跟老齐学 Python 数据分析

齐伟 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

内 容 简 介

读者在本书中可以学习到与数据分析、机器学习相关的 Python 库的应用，并通过各种类型的应用示例将所学基本知识进行综合应用。

本书依然秉承“跟老齐学 Python”系列书的写作风格，力争以通俗易懂的内容与读者分享笔者的心得。虽然数据分析强调的是严谨的科学性和缜密的逻辑性，但本书并不会因为顾此特点而变得枯燥。

本书可作为数据分析工程师、机器学习工程师的入门教程。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

跟老齐学 Python：数据分析 / 齐伟编著. —北京：电子工业出版社，2018.6
ISBN 978-7-121-34003-1

Ⅰ. ①跟… Ⅱ. ①齐… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字（2018）第 070382 号

策划编辑：高洪霞
责任编辑：牛 勇
印 刷：三河市良远印务有限公司
装 订：三河市良远印务有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编：100036
开 本：787×1092 1/16 印张：20.75 字数：541 千字
版 次：2018 年 6 月第 1 版
印 次：2018 年 6 月第 1 次印刷
定 价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

序

认真阅读序言，是读书的好习惯。

“跟老齐学 Python”系列已经出版了三本书，第一本是 Python 入门教程——《跟老齐学 Python：轻松入门》，第二本是 Web 开发教程——《跟老齐学 Python：Django 实战》，本书是第三本。在阅读本书之前，需要读者完成《跟老齐学 Python：轻松入门》的学习或者具有相当程度的知识。

我遇到的学生、软件工程师、大学教师等，他们以各种理由来说明在数据分析、机器学习中应该选择哪种语言或者什么工具。在实际工作中，也的确是百花齐放、百家争鸣。那么到底学什么呢？

这是一个令人苦恼的问题，也是一个浪费时间的问题。

幸亏，现实给出了一个统计性的答案：Python 已经胜出。

所以，本书就呈现在读者眼前了。

本书的目标如下。

- 数据分析和机器学习的入门读物。凡是有志于在此领域工作的读者，通过阅读本书，能够跨进该领域，为日后工作奠定基础。
- 在示例中学习。本书在适当时机，向读者提供各种类型的示例（因为面向对象中的“实例”有特别含义，所以本书中用“示例”表示“某知识技能的应用举例”）。
- 为读者展示一种学习方法。这也是我在前面两本书中所贯彻的核心思想，本书继承这个思想。

本书冠名“数据分析”，是因为绝大部分内容介绍了数据分析的知识和应用，读者学习完这些内容，即可从事相关的工作。在本书的最后一章，也以示例的方式简要介绍了机器学习中的一点内容，主要目的是“开个天窗”，让具有数据分析知识的读者能够看到更广阔的天空。当然，也夹带了私活，就是预告系列丛书的下一本“机器学习”。

跟本书有关的网址如下。

- 代码仓库：https://github.com/qiwsir/DataAnalysis
- 网站：http://itdiffer.com/

如果本书能够成为读者进入数据分析、机器学习领域的垫脚石，我当荣幸之至。

这本书的编写完全是在业余时间完成的，所幸有妻子相助，感谢我的妻子，她为我的写作提供了很多帮助，除日常生活外，还协助我查询和翻译一些资料，通读了全书内容，修正了很多语言表达方面的错误，比如语法错误、错别字等。

另外，还要感谢本书的编辑朋友们，正是有了她们细致、耐心的工作，才能够让本书呈现在读者面前。

齐　伟

2018 年 3 月

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/34003*

目录

第 0 章　数据分析概述 …… 1

0.1　与数据相关的概念 …… 1
0.2　数据分析技术的发展 …… 3
0.3　开发环境配置 …… 5

第 1 章　NumPy 基础和应用 …… 9

1.1　数组对象基础 …… 9
1.2　数组的索引和切片 …… 25
1.3　针对数组的操作 …… 36
1.4　运算和通用函数 …… 46
1.5　简单统计应用 …… 53
1.6　矩阵 …… 57
1.7　矢量运算 …… 60
1.8　综合应用示例 …… 68

第 2 章　Pandas 基础和应用 …… 75

2.1　常用数据对象 …… 75
2.2　索引对象 …… 88
2.3　数据索引和切片 …… 95
2.4　文件读写操作 …… 107
2.5　处理缺失数据 …… 116
2.6　规整数据 …… 121
2.7　分组运算 …… 141
2.8　矢量化字符串 …… 158
2.9　与时间相关的操作 …… 161
2.10　简单的应用示例 …… 174

第 3 章　数据可视化 …… 179

3.1　Matplotlib 概览 …… 179
3.2　设置坐标系 …… 186

3.3 绘制图像……197
3.4 常用统计图……211
3.5 绘制三维图像……225
3.6 Seaborn 掠影……231
第 4 章 综合应用……235
4.1 分析股票数据……235
4.2 分析文胸评论数据……245
4.3 分析电影票房数据……249
4.4 可视化城市人口数据……253
4.5 分析希腊葡萄酒数据……259
4.6 应用本福特定律……273
4.7 制作词云……278
第 5 章 机器学习……283
5.1 线性回归……283
5.2 线性回归示例……299
5.3 Logistic 回归……304
5.4 贝叶斯方法……314
跋……324

第0章

数据分析概述

近几年，数据越来越受到重视了，各种跟数据有关的概念不断被“热炒”。但是，当嘴里不断说出带有“数据”的句子时，是不是也同时明晰了其真正含义？我们是人云亦云，还是基于对其深入研究后的理智思考？

从现在开始，我们共同学习数据分析，厘清各种概念，掌握应用方法，并且透过数据“把这纷扰看个清清楚楚明明白白真真切切”。

0.1 与数据相关的概念

在任何一个学科门类中，概念都非常重要。通过概念能够准确地说明所指对象，概念是我们准确表达所不可或缺的。数据分析及其相关领域正在蓬勃发展，各种新概念层出不穷。为了让读者不至于在面对纷繁多变的概念时惊慌失措，本节将数据分析及其相关领域的几个主要概念罗列出来，请读者阅读，特别是不愿意被名词忽悠的读者，一定要认真阅读这些概念的说明，虽然它们很枯燥，读起来不舒服。不过有“鸡汤”提供者说，越和自己的不舒服做斗争，越能高于一般人。

1．数据

数据，英文是 Data。首先要注意不是“数、数字、数值”，虽然这些可以是数据。《维基百科》中的“数据”词条解释为“未经过处理的原始记录”。

由此可知，数据首先是“记录”，用某种方式把某对象保存下来，这就是数据。比如走进餐厅，用录音器材记录人们吃饭时的声音，这就是声音数据；用摄像器材拍摄公路上个别存在的开车违章现象，这就是影像数据。

再者，定义中所言“未经处理”，是说明这种数据缺乏有效组织，并没有被加工成某种序列信息。

或许，这只是对数据的诸多定义中的一种，仅供参考。

如果仅仅按照上面的方式理解，未免太不“计算机”了。在计算机行业，认为数据是能够

被计算机识别、存储和加工处理的。这虽然不是概念定义，但由此读者理解了软件工程师要处理的数据应当具备的属性。

2．大数据

表述对某个物体的某种测量结果，常常用“数值+单位”的模式。在国际基本单位中，长度的单位是“米”，为了便于应用，在此基础上扩展了其他单位，比如“千米”，更大的单位如“光年”。

在计算机行业，数据的单位有比特（bit），比这个大一点的单位是字节（Byte，1 字节=8 比特）。此外，还有下面各种单位。

KB (Kilo Byte) : 1KB = 2^{10} Byte = 1024 Bytes
MB (Mega Byte) : 1MB = 2^{20} Byte = 1024 KB
GB (Giga Byte) : 1GB = 2^{30} Byte = 1024 MB
TB (Tera Byte) : 1TB = 2^{40} Byte = 1024 GB

打开计算机，通过查看文件属性，能直观地看到每个文件中数据量的多少。这几年，随着网络的发展，某些数据的数值太大了（也可以说数据量很大）。比如，据说 Facebook 每天增加的数据量超过 500TB（消息来源：http://www.infoq.com/cn/news/2012/08/FB-collect-500TB-everyday）。笔者有一个 3GB 的 U 盘，如果用这种 U 盘来存储 Facebook 每天增加的数据，需要多少个这样的 U 盘？

$500 \times 1024 / 3 = 170666.66666666666$

另外一个数据也挺吓人的。据传截至 2012 年，全世界每天产生 2.5 艾字节（2.5×10^{18} 字节）的数据（消息来源：https://www.ibm.com/big-data/us/en/），还用笔者那种 U 盘来存储，需要多少个？

$2.5\times10^{18}/(2^{30}\times3) = 776102145.5128988$

突然想起在 20 世纪 90 年代，笔者还在使用一种容量大约是 1.44MB 的 3.5 英寸的软盘。

可见，不同的对象，其数据大小不同，特别是随着数据量比较大的数据越来越多，很多资料上开始出现了“大数据”这个词语。所谓“大”，应该就是相对某种“小”而言的，这种在“数据”之前添加一个形容词的方法，某种程度上是为了吸引眼球。至少对于“大数据”而言，并不构成一个独立的严谨概念，它只是用来指代某种数据量比较大的对象。但是大于哪一个数值才算“大”呢？没有严格定义。

尽管如此，我们还会面对“大量数据”，此时对技术也有了新的要求，比如数据的存储、数据分析算法等，所以也常常将这些技术方法用“大数据”来指代。这就让本来含糊不清的“大数据”更笼统了，貌似是一个筐——什么都能装。所幸本书的目的不是进行严格的学术名词界定，也就不对“大数据”进行深究，只是从俗使用罢了。

3．数据分析

数据分析，顾名思义，就是通过分析数据，得到某种结果。被分析的数据对象可“小”、可“大”。

假想一个可能荒唐的场景。我们的祖先，从树上到地面的时间还不是很长的时候，掌握了从 1 到 10 这几个数字（是不是通过观察双手习得的呢）。笔者基本可以保证他们那时的计数方法不是从 0 开始的。某天，祖先们经过长途跋涉，到了一个有食物的地方，带头大姐（注意，

那时是母系社会）清点了一下人数，数了好几遍后终于确定一共 9 个人。通过这个数据，她得出一个结论：这次迁徙比较幸运，一个成员也没少。

这是否算是最早的“数据分析”？如果算，那么数据分析与我们形影不离。

事实上，现在所说的数据分析不是这样简单的过程。

一般认为，数据分析是通过某种方法，从不同维度提炼出数据中所包含的信息。用于分析的方法可以概括为以下几个。

- 描述性分析法（Descriptive Analytics）。比如每个月所挣的工钱按照如下方式分配：房租 40%，食品 30%，交通 20%，通信 10%。也可用饼图等图示直观地表示这种分配方式。
- 预测性分析法（Predictive Analytics）。最典型的就是对股票数据的分析，通过分析结果预测买哪只股票会赚钱。“大数据”通常被认为可以很好地预测未来——准确地说应该是在统计意义上预测。
- 规范性分析（Prescriptive Analytics）。先重复一遍或许读者已经知道的“啤酒与尿布”的故事。据传在 20 世纪 90 年代，美国沃尔玛超市的管理人员分析销售数据时发现，在某些特定的情况下，“啤酒”与“尿布”两件看上去毫无关系的商品会经常出现在同一个购物篮中（对其原因的解释，读者可以上网搜索）。根据此分析结果，超市管理人员开始在卖场尝试将啤酒与尿布摆放在相同的区域。这就是非常典型的所谓“数据驱动”的决策。

或许现实中的“数据分析”不能简单地贴上某一个标签，但从上面三个类别中，也能初步理解数据分析的大体用途和方法。

虽然数据分析的数学基础早在 20 世纪 90 年代已经确立，但直到计算机发展起来之后，伴随着计算能力的提升和数据量的增加，“数据科学”才成为人们关注的焦点。可以说，它是数学与计算机科学相结合的产物。

另外，还有一个类似的名词“数据挖掘（Data Mining）”，有人专门撰文区分两者，也有不少资料将两者混用，还有资料只说其一，不提另外一个，言下之意是两者一样，用一个词语即可。总之，在当前盛产新词汇的时代，面对众多似是而非的名词做取舍，也的确不知所措。所以，笔者的做法就是“不争论”，重实务。

以上列出三个跟数据有关的概念，主要目的在于使读者对“数据”领域有一个概括的认识。而在实际的“数据”语境中，名词绝非上述三个，还有云计算、集群计算、暗数据、数据湖、脏数据、结构化数据、数据科学家、数据分析师、机器学习、人工智能、数据清洗等。或许在本书中读者也会看到一些似是而非、似懂非懂的词语，如果想深究，建议“自己动手，上网搜索”——身处一个造词的新时代，谁也逃不脱。

“透过历史，看××未来”，这是一句常用的话，其中也蕴含着数据分析方法，历史就是过去的数据，需要详查。

0.2　数据分析技术的发展

谁没有历史？！数据分析不是凭空出现的，有一个发展历程，这就是它的历史。了解历史的目的不仅是在茶余饭后聊天用，更是通过历史，在一定程度上判断其发展趋势，并用于今天

和明天的决策之中——这也是数据分析。通过下述的历史过程，或许读者能够判断这门学科的发展趋势，从而确定是否有必要认真阅读本书，是否立志在数据分析领域从业等——风闻，数据分析和机器学习的工程师都是高薪哦。

下面就是数据分析的极简编年史（参考 Gil Press 的 *A Very Short History Of Data Science*，网址是 https://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/#5c5686a355cf）。

- 1947 年，John W. Tukey 创造了“比特”这个词。
- 1948 年，Claude Shannon 在论文《通信的数学理论》中使用了“比特”一词。
- 1962 年，John W. Tukey 在《数据分析的未来》一文中提出“数据分析本质上是一门实证科学”。1977 年，他又发表了论文《探索性数据分析》。他认为需要更加重视利用数据来检验所提出的假设，并进一步提出“探索性数据分析”和“验证性数据分析”“可以且应该并行”。
- 1974 年，Peter Naur 出版了《计算机方法的简明综述》，这本书综述了被广泛应用的当代数据处理方法。
- 1989 年，Gregory Piatetsky-Shapiro 组织和主持了第一届 KDD（Knowledge Discover in Databases）研讨会。1995 年，KDD 研讨会成为 ACM SIGKDD 年度会议。
- 1994 年 9 月，《商业周刊》发表了关于“数据库营销”的封面故事，“公司正在收集关于你的大量信息，对这些数据信息进行运算，从而预测你购买某一种产品的可能性，并利用这些知识来制定出精确校准过的营销信息来促使你购买它”。
- 1996 年，Usama Fayyad、Gregory Piatetsky-Shapiro 和 Padhraic Smyth 发表了《从数据挖掘到数据库中的知识发现》，他们认为“KDD 是指从数据中发现有用的知识的全过程，而数据挖掘是指在这一过程中的具体步骤”。
- 1997 年，C. F. Jeff Wu 教授在密歇根大学的演说中呼吁将统计学改名为数据科学，并且将统计学家改名为数据科学家。
- 2002 年 4 月，《数据科学杂志》创刊。
- 2005 年 5 月，Thomas H. Davenport、Don Cohen 和 Al Jacobson 发表研究报告《分析学方面的竞争》，描述了一种新的竞争形式，即统计、定量分析及预测模型开始代替传统因素成为公司竞争的主要元素。这项研究后来被 Davenport 发表于《哈佛商业评论》（2006 年 1 月），之后他同 Jeanne G. Harris 一起将其研究成果编成图书《分析学方面的竞争：致胜的新科学》（2007 年 3 月）。
- 2007 年，数据科学研究中心在复旦大学建立。
- 2009 年 1 月，题为《利用数字数据的力量服务于科学和社会》的报告出版。报告指出，“许多学科中涌现出一种新型的数据科学和管理专家，他们擅长计算机、信息、数据科学领域及另一个科学领域。这些人是当前和未来科研事业成功的关键”。
- 2009 年 1 月，谷歌首席经济学家 Hal Varian 告诉麦肯锡季刊：“掌握数据的能力——能够理解数据、处理数据，从中提取有价值的信息，把数据可视化，传达数据——这将是未来几十年的一项非常重要的技能……因为现在我们确实拥有基本上免费和无处不在的数据。因此，理解这些数据并从中获取价值的能力成为稀缺因素……我的确认为这些技能——能够获取、理解和传达你从数据分析中所获得的见解——将是非常重要的。管

理人员需要能够访问和理解数据本身”。

上述简史终止于 2009 年，虽然那年之后，围绕数据发生的事情还有很多，并且发展迅猛，但是笔者认为 Hal Varian 已经做了很好的预见，后面的事情都在证明他的预见。

“滚滚长江东逝水，浪花淘尽英雄”，众多“牛人”们为我们今天的学习奠定了基础，是他们给了我们可以依靠并站立于上的肩膀，所以我们要“时刻准备着”，为“数据分析的伟大事业”贡献自己绵薄的力量（此处参考了小学作文的部分语句，请回忆）。

0.3　开发环境配置

本节讲解数据分析的工具和使用方法——以 Python 语言为基础的库。在现实中，能够用于数据分析的工具五花八门，概括起来，可以分为以下两大类。

- GUI 软件产品，比如 SPSS、电子表格等，这种工具操作简单，获得了不少用户的青睐。但是由于它们自身是一款软件产品，只能提供软件产品本身的功能，对于“大数据”中多样化的需求就显得无能为力了。还有一个问题也需要提醒读者，它们不都是免费的。很多从事数据分析方面研究、教学和学习的用户，以及使用数据分析工具的用户，或许应用的是盗版软件，对此笔者旗帜鲜明地反对。
- 以 Python 和 R 为代表的高级编程语言，在数据分析领域已经被广泛使用，特别是在处理“大数据”时，其优势不仅在于开源免费，更重要的是能够根据业务需要，灵活多样地进行各种计算，并且在所需要的方向上进行优化。本书当然选择 Python，因为它是“跟老齐学 Python”系列图书中的一员。更重要的是，现实已经表明，Python 在数据科学领域的霸主地位已经确立，它是每个试图进入此领域的人不得不学的——所以《跟老齐学 Python：轻松入门》之后要继续《跟老齐学 Python：数据分析》。

使用 Python 进行数据分析，不是用鼠标点来点去就能解决的，必须使用一些专门的第三方库。所以，安装相应的库是必需的。

笔者默认阅读本书的读者已经完成《跟老齐学 Python：轻松入门》的学习，或者具有相当的 Python 知识技能。

在本书中，通常都是使用 pip 命令安装有关数据分析的库，并且针对 Python 3。

尽管以下要安装的库已经成为数据分析的标配，但是至今它们也没有被纳入标准库，所以要自己安装。

本书所有代码都是基于 Ubuntu 16.04 操作系统调试的——笔者不厌其烦地在各种适合的场合推荐的操作系统。

1．安装基本库

打开终端，依次输入如下安装指令，特别建议一个一个进行安装。根据经验，有的库可能由于连接超时而安装失败，遇到这种情况笔者也无能为力，除非改变安装方式。有的库安装时间比较长，要有耐心等待。

```
$ sudo pip3 install numpy
$ sudo pip3 install scipy
$ sudo pip3 install matplotlib
```

```
$ sudo pip3 install pandas
$ sudo pip3 install sympy
$ sudo pip3 install ipython
$ sudo pip3 install jupyter
```

安装完毕，会显示是否成功。若还不放心，可以用类似下述方法检验。

```
$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>>
```

import 后面的那个库，如果不报错，就说明安装成功。

以上安装的仅仅是基础设施，在学习过程中，还会根据需要安装别的库。

因为上述所安装的库都是开源的，所以能够将它们集结在一个大包中，这就是所谓集成式的安装包。通过这个安装包，就能获得上述所有库的环境。其中比较有名的是 Anaconda，用官方网站的说明，它是“The Most Popular Data Science Ecosystem”。下载之后，一次安装，获得上述要求的配置（其实比上述还多）。下载地址是 https://www.continuum.io/downloads，分别有 Windows、Mac OS、Linux 三种操作系统的安装包供下载使用。

诚然，跟其他开源软件一样，还可以下载源码编译安装，这种方式不是笔者在这里推荐的。喜欢用这种方式安装的读者，一定也是高手，笔者就不演示了。

如果读者穷尽所能，依然无法将所要求的环境配置好，那么只能求助搜索引擎了。为此特别奉上一句话：

“搜索决定成败”——老齐

推荐的搜索引擎当然是 google.com，不管你有什么理由，但凡想在数据分析和机器学习领域从业，必须用它——虽然它更多时候是图 0-3-1 这样的，但为了你的目标，也要努力去寻找。

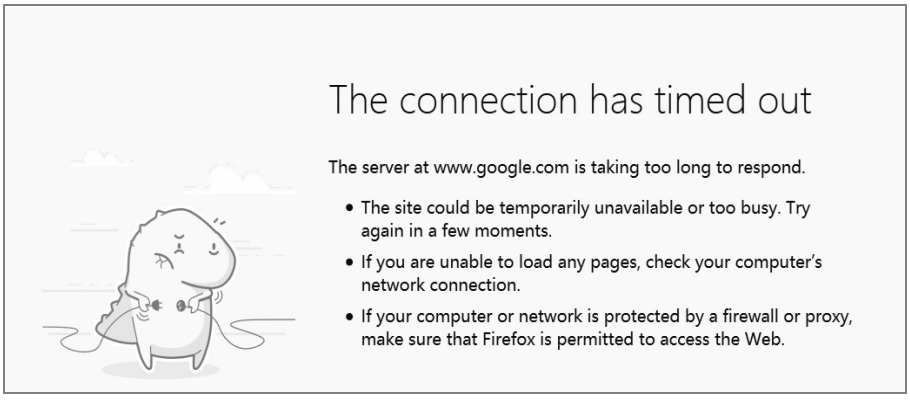


图 0-3-1　google.com 连接超时

准备工作完成，就要开始学习了。

2．使用 Jupyter

在学习 Python 的时候，我们经常在交互模式中进行一些操作。但是，Python 默认的交互模式其实很不友好，你不觉得吗？有这种感觉的人应该不少吧。所以，Jupyter 横空出世了。

Jupyter 官方网站的网址是 http://jupyter.org/，如图 0-3-2 所示。首页这样描述：“Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages”。

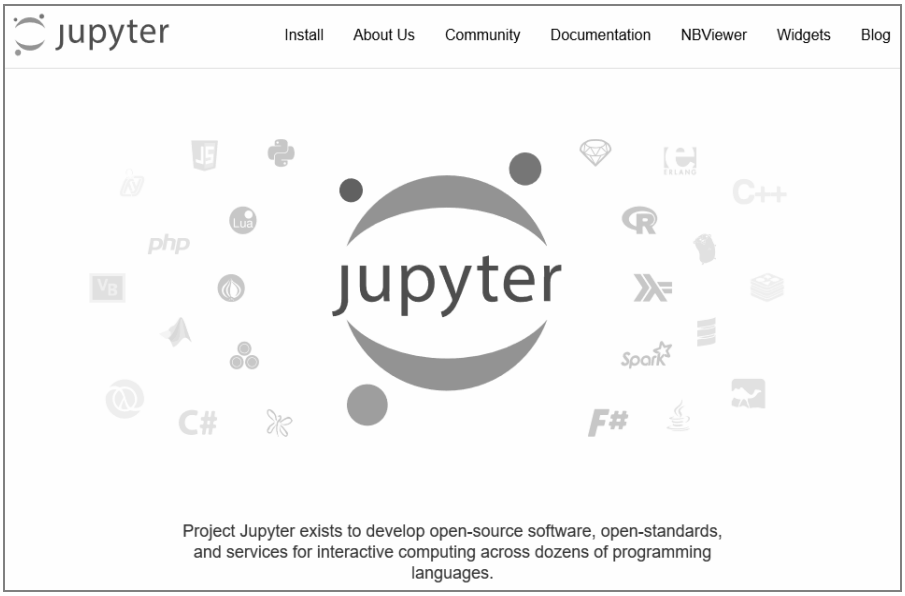


图 0-3-2　Jupyter 官网

Jupyter Notebook 是一种基于浏览器的交互环境，支持的不仅有 Python，也有别的语言。读者在有的资料里还会看到 IPython Notebook，这是它的曾用名。

执行如下命令。

```
qiwsir@ubuntu:~$ jupyter notebook
[I 14:28:46.797 NotebookApp] Serving notebooks from local directory: /home/qiwsir
[I 14:28:46.797 NotebookApp] 0 active kernels
[I 14:28:46.797 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/?token=b87a995704e95d9b7ca653e4065aa3c380c73f0aa3a8dd16
[I 14:28:46.797 NotebookApp] Use Control-C to stop this server and shut down all kernels
(twice to skip confirmation).
[C 14:28:46.801 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:8888/?token=b87a995704e95d9b7ca653e4065aa3c380c73f0aa3a8dd16
[I 14:28:51.780 NotebookApp] Accepting one-time-token-authenticated connection from
127.0.0.1
```

执行上述命令后会自动打开默认浏览器，显示类似如图 0-3-3 所示的界面。

图 0-3-3　启动 Jupyter Notebook

单击图 0-3-3 所示界面中的 New 下拉按钮，在下拉菜单中选择 Python 3，如图 0-3-4 所示。

图 0-3-4　选择 Python 3 交互界面

这时会创建一个新的 Tab，这就是我们的工作界面，如图 0-3-5 所示。

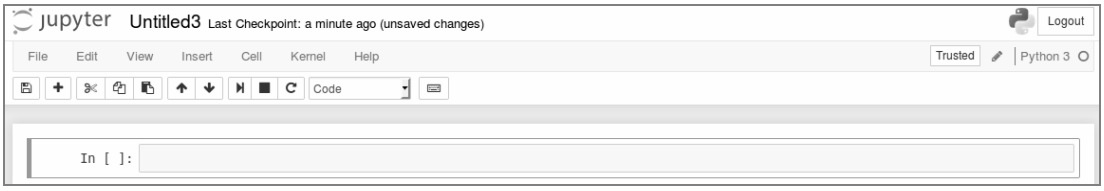


图 0-3-5　Jupyter Notebook 工作界面

关于 Jupyter 的一些使用技巧不是本书的重点，建议读者上网查找资料了解。它的操作非常简单，读者根据已有的软件操作经验，操作它肯定不在话下。

在 Jupyter Notebook 中操作，最终可以将当前页面上的内容保存为扩展名是.ipynb 的文件，这个文件可以传播，也可以将其他的这种文件导入到当前的 Jupyter Notebook 界面中。比如，读者可以在本书代码仓库中下载相应文件并导入——其实笔者不提倡，笔者更提倡自己敲代码。

万事俱备只欠东风，还有你的决心——开始练“神功”。

第1章

NumPy基础和应用

NumPy 是 Python 语言的一个第三方库，被广泛应用于数据分析领域。它实现了多维数组与矩阵的高效运算，还提供了大量的数学函数。用更高、更快、更强来描述 NumPy 并不为过，“更高”即开发效率高，“更快”即运行速度快，“更强”即数据处理方面的功能强大。毫不夸张地说，NumPy 是任何打算进入数据分析乃至机器学习、人工智能等领域的读者必须要学习并掌握的。

NumPy 的前身是一款名为 Numeric 的库，由 Jim Hugunin 与其他协作者共同开发。2005 年，Travis Oliphant 在 Numeric 中结合另一个同性质的库 Numarray 的特点，并加入了其他扩展而开发了 NumPy。

NumPy 是开源的，其最大的好处就是免费，因此其代码质量能够得到最大限度地保证——开源，意味着最大限度的安全。

NumPy 是数据分析“降龙十八掌”的第一招。

1.1 数组对象基础

“ndarray”是 NumPy 的核心功能，其含义为 n-dimensional array，即多维数组。在后面的叙述中，会经常用到“数组”这个词，就是指的 ndarray。数组是 NumPy 的一个重要数据结构，正如 Python 中“万物皆对象”原则，数组也是一个对象，这个对象具有自身的独特之处，具体表现在其属性和方法上。

1．初识数组对象

（1）使用 Jupyter Notebook

如果读者按部就班地跟随本书操作，那么已经打开了 Jupyter Notebook 界面。

执行如下操作。

```
In [1]: import numpy as np
        np.__version__
```

```
Out[1]: '1.13.0'
```

In[1]表示输入的内容。输入完毕，按住 Shift 键，再按 Enter 键，就会执行输入语句。如果有结果出来，就会在 Out[1]中显示。In[1]中的数字是程序单元的序号。看一下笔者执行 In[1]的截图，如图 1-1-1 所示。

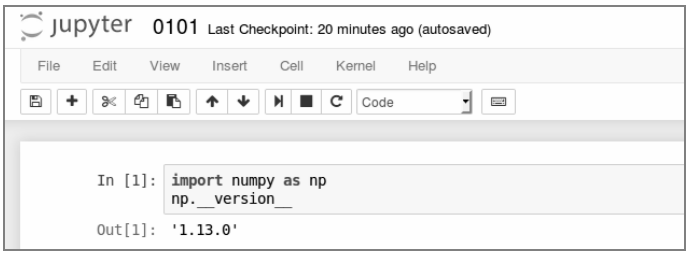


图 1-1-1　执行结果图示

Out[1]的输出结果是当前 NumPy 的版本。或许读者所使用的版本跟笔者所演示的不同，这是很正常的。因为 NumPy 一直在发展，这也是我们使用它的重要原因。如果它的版本号不变化了，你还敢用吗？这又不是古董。当然，新旧版本会有一些差异，但读者对此不必太担心。一方面主体内容不会有太大变化（除非比较大的版本变化，比如升级为 2.xx.x），另一方面笔者在本书中各章节重点强调的不是掌握哪些知识，而是要掌握学习知识的方法，本书中的知识只不过是方法的载体罢了。再者，NumPy 有非常好的帮助文档，甚至在操作的时候，都会有非常友好的提示。

在 In[1]中，就是引入 NumPy。通常都用这种方式引入，请读者也用这种通常的方式——要获得自由，必须遵守规范——这种引入方式能够保证你跟别人快乐地一起玩耍，不至于弄翻友谊的小船。

为了认识数组对象，先要创建一个数组。下面就创建本书的第一个数组。

```
In [2]: data = np.array([1, 2, 3, 4, 5])
        data
Out[2]: array([1, 2, 3, 4, 5])

In [3]: type(data)
Out[3]: numpy.ndarray
```

关于如何创建数组，后面会专门讲解，这里先来认识一下数组。

刚才所创建的数组，其类型为 numpy.ndarray，这个数组即为一个对象。既然如此，它就有一些属性和方法，所以 dir()在这里依然有效（如果读者不知道 dir()的作用，请查阅《跟老齐学 Python：轻松入门》）。

```
In [4]: dir(data)
Out[4]: ['T',
         '__abs__',
         '__add__',
         ……                    #省略很多内容
         'transpose',
         'var',
         'view']
```

从 Out[4]的输出列表中，读者可以看到很多数组对象的属性和方法。

除使用 dir()外，还有另外一个查看当前对象属性和方法的操作方式。

```
In [5]: data?
```

此时会显示如图 1-1-2 所示的内容。

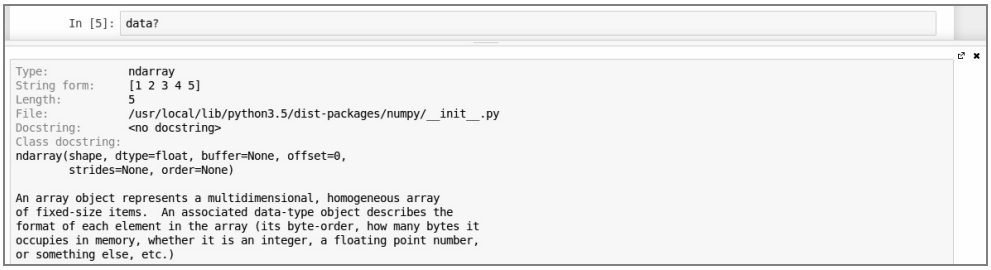


图 1-1-2　data 帮助文档部分截图

单击帮助文档界面右上角的两个小图标，可分别实现在新 Tab 中全屏查看和关闭当前文档这两个功能。

通过 In[5]看到的内容，是数组的官方文档，对数组对象做了详细说明，其中蕴含的信息非常丰富。如果读者有耐心，特别建议将上述内容进行详细浏览，即使不理解，也能够对数组对象有初步的印象，再结合本书的叙述，会对学习大有裨益。请特别留意 In[5]使用的方法，这是本书提倡的学习方法。

（2）数组元素的类型

In[2]所创建的数组对象中的数字（1, 2, 3, 4, 5）称为数组的元素，它们是数组这个数据结构中的具体内容。

```
In [6]: data.dtype
Out[6]: dtype('int64')
```

通过数组对象的 dtype 属性可以得到组成数组的元素的类型。在 In[6]的操作中，可以得知数组 data 的元素类型是 int64，即 64 位整数。

数组的元素都有哪些类型？

在回答这个问题之前，必须要牢记：组成数组的元素必须是同一种类型。

这显然跟 Python 中的列表不同（建议读者在学习数组的同时，不断跟以往的知识进行对比，特别是 Python 的列表）。

为了适应 NumPy 所要担负的计算工作，数组中元素的内置类型（已经定义好的类型，除此之外，还可以自定义元素的类型）主要有以下几种，如表 1-1-1 所示。

表 1-1-1　NumPy 内置的数组元素类型

类　　型	字符编码	说　　明
int：int8，int16，int32，int64	i：i1，i2，i4，i8	有符号整数型。int8 表示其长度为 8 位（1 字节），能够表示−128～127 内的整数，其他类型依此类推
uint：uint8，uint16，uint32，uint64	u：u1，u2，u4，u8	无符号整数型。位数含义同上

续表

类　型	字符编码	说　明
bool	b	布尔型
float：float16，float32，float64，float128	f2，f4/f，f8/d，f16/g	浮点数。其中 float16 为半精度浮点数，float32 为单精度浮点数，float64 为双精度浮点数，float128 为扩展精度浮点数
complex：complex64，complex128，complex256	D：c8，c16，c32	复数。分别用 32 位、64 位或 128 位表示复数的实部和虚部
string_	S	固定长度的字符串类型
unicode_	U	固定长度的 unicode 类型

表 1-1-1 中的“字符编码”是每种类型对应的字符串的表示方法，这是为了兼容 NumPy 的前身 Numeric 而设定的。虽然现在使用得越来越少了，但有的程序中还会出现，读者遇到时可以查阅。

某个数组一旦建立，其元素类型是确定不能随意改变的。那么，在实际业务中如果遇到需要修改元素类型的情况，应该如何操作？

```
In [7]: new_data = data.astype(np.float)
        new_data
Out[7]: array([ 1.,  2.,  3.,  4.,  5.])

In [8]: new_data.dtype
Out[8]: dtype('float64')

In [9]: data, data.dtype
Out[9]: (array([1, 2, 3, 4, 5]), dtype('int64'))
```

astype()是数组对象的一个方法，它能够根据指定的类型（参数指定元素类型）新生成一个数组，新数组 new_data 相对于旧数组 data 只有元素的类型不同。

（3）数组的外貌

除 dtype 属性外，数组对象还有其他一些常用的属性，通过这些属性能够了解数组对象的基本概况，比如数组的形状、维度等。

```
In [10]: a = np.array([1, 2, 3])
         b = np.array([1.0, 2.0, 3.0])
         a.dtype, b.dtype
Out[10]: (dtype('int64'), dtype('float64'))
```

In [10]创建了 a、b 两个数组，这两个数组的差别在于创建的时候，列表中的元素类型不同。那么，对两个数组分别输出 dtype 属性，得到的结果也不同，这是显然的。但从外形上看，这两个数组还是有相同之处的。

```
In [11]: a.shape
Out[11]: (3,)

In [12]: b.shape
Out[12]: (3,)
```

shape 的英文含义是“形状”，这个属性返回的是一个元组（tuple），其中的元素是由整数构

成的，它们代表着该数组的形状——每个轴的元素数（关于“轴”的概念，请见后续讲解），“(3,)”的含义是 0 轴上有 3 个元素。

```
In [13]: c = np.array([1.0, 2.0, 3.0, 4.0])
         c.shape
Out[13]: (4,)
```

In[13]创建了一个数组 c，它的形状跟前面的数组 a、b 不同，shape 属性的返回值为(4,)，即数组 c 的 0 轴上有 4 个元素。不管数组 a 还是数组 c，尽管两个数组在 0 轴上的元素数不同，但它们都只有一个轴，即 0 轴，这种只有一个轴的数组被称为一维数组。对此，也有一个属性，返回数组的维度。

```
In [14]: a.ndim
Out[14]: 1
```

观察 a、c 两个数组，它们的元素个数不同。数组对象还有一个专门返回元素个数的属性 size，相关演示如下。

```
In [15]: a.size
Out[15]: 3

In [16]: c.size
Out[16]: 4
```

NumPy 数组对象的常用属性如表 1-1-2 所示，请读者自行操作测试。

表 1-1-2　NumPy 数组对象的常用属性

属　性	说　明
dtype	返回数组中元素的类型
shape	返回由整数组成的元组，元组中的每个整数依次对应数组的每个轴的元素个数
size	返回一个整数，代表数组中元素的个数
ndim	返回一个整数，代表数组的轴的个数，即维度
nbytes	返回一个整数，代表用于保存数据的字节数

在 Python 中，当我们需要求助于文档的时候，就会使用 help()，这种方法在这里依然可以使用，比如 help(a.dtype)。当然，在 Jupyter 中，还可以使用“a.dtype?”查看文档，建议读者试一试。

2．创建数组

虽然已经对数据有了一点点了解，但终究是走马观花地看了看外貌。好比找伴侣，外貌固然重要，但门当户对也要强调，所以要看出身。数组的出身，就源于其创建方法。

（1）创建数组的基本方法

np.array()是创建数组的基本方法，虽然前面已经使用了这个方法，但为了窥其全貌，还要看一看完整的文档说明。

```
In [17]: np.array?
Docstring:
array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)
```

```
Create an array.

Parameters
----------
```

- object : array_like

  An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

- dtype : data-type, optional

  The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.

- copy : bool, optional

  If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (`dtype`, `order`, etc.).

- order : {'K', 'A', 'C', 'F'}, optional

  Specify the memory layout of the array. If object is not an array, the newly created array will be in C order (row major) unless 'F' is specified, in which case it will be in Fortran order (column major). If object is an array the following holds.

```
    ===== ========= ====================================================
    order  no copy                     copy=True
    ===== ========= ====================================================
    'K'   unchanged F & C order preserved, otherwise most similar order
    'A'   unchanged F order if input is F and not C, otherwise C order
    'C'   C order   C order
    'F'   F order   F order
    ===== ========= ====================================================
```

```
  When ``copy=False`` and a copy is made for other reasons, the result is the same as if ``copy=True``, with some exceptions for `A`, see the Notes section. The default order is 'K'.
```

- subok : bool, optional

  If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

- ndmin : int, optional

  Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement.

```
Returns
-------
out : ndarray
    An array object satisfying the specified requirements.
```

因为篇幅所限，这里仅列出文档内容的一部分，并且为了便于阅读，进行了重新排版。

能够耐心地阅读文档，是一项必要的修炼，也是笔者特别强调的。如果读者已经明白此道理并可以实行，必将掌握本书的要诀。在这里演示这种重要的学习方法，后续可能就一带而过，不再演示了，但并不意味着不重要。

下面通过若干示例，说明用 np.array()创建数组的方法——这是基本方法。

```
In [18]: a = np.array([1, 2, 3, 4])    #①
         b = np.array([1, 2, 3, 4], dtype=float)    ②
         a
Out[18]: array([1, 2, 3, 4])

In [19]: a.dtype
Out[19]: dtype('int64')

In [20]: b
Out[20]: array([ 1.,  2.,  3.,  4.])

In [21]: b.dtype
Out[21]: dtype('float64')
```

比较 In[18]中①和②两种创建数组的方式，向 array()传入的参数都是[1, 2, 3, 4]，差别在于②中声明了元素的类型（dtype=float），分别用 a、b 两个数组的 dtype 属性查看其元素类型（如 Out[19]和 Out[21]所示），数组 a 的元素类型是整数，数组 b 的元素类型为浮点数。

在 Python 的列表中，我们已经了解了嵌套列表，即列表中的元素还是列表。在创建数组的时候，如果我们传入的是嵌套列表，则得到的就是一个多维数组，来看示例。

```
In [22]: da = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
         da
Out[22]: array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 9, 10, 11, 12]])
```

在 In[22]中向 np.array()传入了[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]，注意这个嵌套列表的各个元素的长度必须一样。同时，因为数组的元素类型必须一致，所以列表中的元素类型也必须相同（如果没有类型的特别声明，NumPy 会自行推断元素类型。比如上述列表中如果有某个数是浮点数，则 NumPy 会将该数组元素类型设定为浮点数）。如果元素参差不齐，也不会报错，只不过得不到我们预想的数组对象。

```
In [23]: np.array([[1, 2, 3], [5, 6, 7, 8], [11, 12]])
Out[23]: array([list([1, 2, 3]), list([5, 6, 7, 8]), list([11, 12])], dtype=object)
```

虽然 In[23]也建立了数组，但从 Out[23]中可以看出，与 Out[22]的结果大不一样，Out[22]才是我们将来会在数据分析中用到的所谓的二维数组。

```
In [24]: da.shape
Out[24]: (3, 4)

In [25]: da.size
Out[25]: 12

In [26]: da.ndim
Out[26]: 2

In [27]: a.shape    #In[18]建立的数组 a = np.array([1, 2, 3, 4])
Out[27]: (4,)
```

```
In [28]: a.size
Out[28]: 4

In [29]: a.ndim
Out[29]: 1
```

In[18]中所创建的数组 a 是一维数组，In[22]中所创建的数组 da 是二维数组，上面的操作分别显示了两个数组的几个属性值。特别要关注 Out[24]的结果，da.shape 的返回值是一个元组，这个元组的第 0 个（注意，在编程语言中，计数都是从 0 开始的）数字“3”表示数组 da 在 0 轴方向上的元素是 3 个，第 1 个数字“4”表示数组 da 在 1 轴方向上的元素是 4 个，整个数组的元素为 3×4 个，即 da.size 的结果（Out[25]）。数组 da 有 0 轴和 1 轴两个轴，即有两个维度，我们就说 da 是二维数组，da.ndim 返回的就是维度数 2，这个值也是 Out[24]的元组的长度。

下面以二维数组为例说明数组的“轴”，如图 1-1-3 所示。

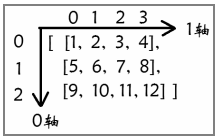


图 1-1-3　二维数组的轴

对于此数组，从外向内看，第一层就是 0 轴，在这个轴方向上有 3 个元素（[1, 2, 3, 4]，[5, 6, 7, 8]，[9, 10, 11, 12]），编号依次为 0、1、2（我们熟知的索引）；再深入第二层，即每个列表元素，每个列表元素都是由 4 个整数构成的，那么这一层就是第二个轴，即 1 轴，这个轴方向上的索引依次是 0、1、2、3。

通过图 1-1-3 可知，0 轴和 1 轴组成了一个二维坐标，数组中的每个元素在这个坐标中都可以用 0 轴和 1 轴的索引唯一锁定，类似我们在数学中学习过的坐标系中的点可以用一组坐标唯一表示。

在对二维数组理解的基础上，也可以建立三维数组、四维数组乃至更多维的数组，读者可以自行尝试一番。

注意，数组中的“维度”，跟我们在数学、物理学科中对空间“维度”的定义是有区别的。比如，我们生活的空间，用物理术语称为“三维空间”，即可以使用由 X、Y、Z 三个坐标轴组成的坐标系描述空间的任何一点。假设在这个坐标系中有一点 A，该点表示为 $A(3, 4, 5)$，并且有一条从坐标原点指向 A 点的有向线段，此即为一个矢量，此矢量可以表示为 $\overrightarrow{OA}=3\vec{i}+4\vec{j}+5\vec{k}$ 。但是，这个矢量如果用数组表示，则为 a = np.array([3, 4, 5])，显然这个数组 a 的维度依然是 1，不是 3。此区别请读者注意。

读者还要注意一点，不要误以为写成一行的就是一维，请看下面的操作。

```
In [30]: db = np.array([1, 2, 3, 4, 5, 6, 7, 8], ndmin=2)
         db
Out[30]: array([[1, 2, 3, 4, 5, 6, 7, 8]])

In [31]: db.shape
Out[31]: (1, 8)
```

```
In [32]: db.ndim
Out[32]: 2
```

仔细观察 Out[30]的输出结果，虽然只有一行，但其本质是一个嵌套，只不过第一层（0 轴）只有一个元素（[1, 2, 3, 4, 5, 6, 7, 8]）。对比操作，更好理解。

```
In [33]: dc = np.array([1, 2, 3, 4, 5, 6, 7, 8])
         dc
Out[33]: array([1, 2, 3, 4, 5, 6, 7, 8])

In [34]: dc.shape
Out[34]: (8,)

In [35]: dc.ndim
Out[35]: 1
```

形成 dc 和 db 两个数组差别的原因在于 In[30]和 In[33]创建数组的时候，ndmin 参数的值不同，其值规定了该数组所应具有的最小维度数。

前面的示例中都是以列表为 np.array()的 object 参数的引用对象，而在其文档说明中很明确地指出 object 参数所引用的对象是 array_like（类数组，可以是列表、元组等可迭代序列，也可以是数组、矩阵），列表是类数组对象，其他符合此条件的对象当然也可以，比如下面的示例。

```
In [36]: a
Out[36]: array([1, 2, 3, 4])

In [37]: de = np.array(a, dtype=complex)
         de
Out[37]: array([ 1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j])

In [38]: de.dtype
Out[38]: dtype('complex128')
```

这次我们直接使用了一个数组，并且将元素类型设置为 dtype=complex，输出结果如 Out[37]所示。如果读者认真阅读了 np.array()的帮助文档，会发现还可以使用矩阵创建数组，矩阵是本书后续要讲解的内容。

使用 np.array()创建数组是一种最基本的方法，此外还有别的方式——不同的方式有不同的用途，都有存在的必要。

（2）用函数创建数组

假设要创建一个数组，它有 100 个整数（也可以更多），并且这些整数具有某种规律，如果还用前面的方法，就要写一个含有 100 个整数（或者更多）的列表，这显然有点不优雅了。程序员是不愿意做这种事情的，这类事情都应该由程序来完成，所以就有了专门用来创建数组的函数，通过专门的函数可以创建有特征的数组。

```
In [39]: np.zeros((2, 10))
Out[39]: array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

np.zeros()能够创建一个完全由 0 组成的数组，(2, 10)是以元组形式声明该数组的形状，即 0

轴元素个数为 2，1 轴元素个数为 10。当然，还有其他参数，读者可以使用“np.zeros?”方法查看文档中的参数列表和说明，下面仅显示来自文档的部分内容，目的是针对性地解释部分参数的含义。

```
zeros(shape, dtype=float, order='C')

Return a new array of given shape and type, filled with zeros.

Parameters
----------
shape : int or sequence of ints
    Shape of the new array, e.g., ``(2, 3)`` or ``2``.
dtype : data-type, optional
    The desired data-type for the array, e.g., `numpy.int8`.  Default is
    `numpy.float64`.
order : {'C', 'F'}, optional
    Whether to store multidimensional data in C- or Fortran-contiguous
    (row- or column-wise) order in memory.
```

请关注一个参数 order='C'，在 np.array()的参数中也有 order。这个参数用于声明本数组元素在数据存储区的排列格式，一般认为有两种格式，一种是根据行排列，另外一种是根据列排列。根据行排列是 C 语言的存储格式，所以在 NumPy 中就称为“C 语言格式”，即 order='C'；根据列排列是 Fortran 语言的存储格式，所以有 order='F'的设置。默认情况下，使用 order='C'。

用于创建数组的函数还有好几个，下面先以表格的形式列出来，再进行说明，如表 1-1-3 所示。

表 1-1-3　NumPy 中创建特殊数组的函数

函　数	说　明
asarray	输入参数为列表、元组，或者由它们组成的嵌套对象或数组，返回一个数组。如果参数是数组，则返回该数组本身
arange	根据开始值、结束值和步长创建一个数组
ones、ones_like	ones 根据指定的形状（以元组方式指定）和元素类型，创建元素值为 1 的数组。ones_like 创建一个与某指定数组完全一样但元素值都为 1 的数组
zeros、zeros_like	与上面雷同，只不过元素值都是 0
empty、empty_like	与上面雷同，只不过没有填充任何元素数据，但分配了内存空间
eye、identity	创建对角线元素是 1、其余元素是 0 的二维数组
diag	创建对角线元素是指定数值、其余元素是 0 的二维数组
linspace	根据开始值、结束值和元素数量创建元素是等差数列的数组
logspace	根据开始值、结束值、元素数量和对数底创建元素是等比数列的数组

为了能够对以上诸函数有所了解，还是建议读者阅读其文档。下面所讲述的内容，纯粹是为了辅助理解文档。要提醒读者注意，这里的说明肯定不如文档中描述的严谨，如有不当之处请以文档所述为准。

① 同一种元素的数组

在 np.zeros((2, 10))中传入了参数(2, 10)，它声明了即将创建的数组的形状，通常用元组表示。

注意观察如下操作。

```
In [40]: np.ones((6,))
Out[40]: array([ 1.,  1.,  1.,  1.,  1.,  1.])

In [41]: np.ones(6)
Out[41]: array([ 1.,  1.,  1.,  1.,  1.,  1.])
```

以上两种不同形式的参数，创建了相同的数组。“(6,)”毫无疑问是一个元组，表示该数组的形状，只有在一维数组的时候，才可以用 6 来替代“(6,)”，但是笔者不推荐这样写，还是严格按照统一规范写代码比较好。

既然 np.zeros()的参数是数组的形状，就有了如下操作。

```
In [42]: da
Out[42]: array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 9, 10, 11, 12]])

In [43]: da.shape
Out[43]: (3, 4)

In [44]: np.ones(da.shape)
Out[44]: array([[ 1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.]])
```

实现上面同样结果的操作，还可以使用 np.ones_like()函数。

```
In [45]: np.ones_like(da)
Out[45]: array([[1, 1, 1, 1],
                [1, 1, 1, 1],
                [1, 1, 1, 1]])

In [46]: np.ones_like(da, dtype=np.float)
Out[46]: array([[ 1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.]])
```

问题只有一个，但解决方法有多个。NumPy 提供了多种函数可以实现同一种操作，以便我们在实践中根据客观需要选用。

利用 np.ones()、np.zeros()、np.empty()得到的是元素为 0、1 或空的特殊数组，那么，是否能得到元素是其他数值的数组呢？

```
In [47]: df = 6.4 * np.ones_like(da)
         df
Out[47]: array([[ 6.4,  6.4,  6.4,  6.4],
                [ 6.4,  6.4,  6.4,  6.4],
                [ 6.4,  6.4,  6.4,  6.4]])
```

In [47]中使用 6.4 这个浮点数乘以元素为 1 的数组，结果是 6.4 与每个元素相乘，最终得到的元素都是 6.4 的数组。像 6.4 这样的数，我们称为“标量”，而后面的数组 np.ones_like(da)则被称为“矢量”。关于运算问题，后面会详细介绍，请保持你的耐心。

```
In [48]: np.full(da.shape, 6.4)
Out[48]: array([[ 6.4,  6.4,  6.4,  6.4],
                [ 6.4,  6.4,  6.4,  6.4],
                [ 6.4,  6.4,  6.4,  6.4]])
```

又看到解决问题的方法不止一种了——真实世界就是这样，方法不是标准的，会有很多选择。

建议读者自行查看 np.full()和 np.full_like()方法的帮助文档。

② 对角线独特的数组

np.eye()、np.identity()和 np.diag()都能够创建对角线元素比较特殊而其他部分的元素为 0 的数组。

```
In [49]: np.eye(4, dtype=int)
Out[49]: array([[1, 0, 0, 0],
                [0, 1, 0, 0],
                [0, 0, 1, 0],
                [0, 0, 0, 1]])

In [50]: np.eye(4, dtype=int, k=1)
Out[50]: array([[0, 1, 0, 0],
                [0, 0, 1, 0],
                [0, 0, 0, 1],
                [0, 0, 0, 0]])

In [51]: np.eye(4, dtype=int, k=-1)
Out[51]: array([[0, 0, 0, 0],
                [1, 0, 0, 0],
                [0, 1, 0, 0],
                [0, 0, 1, 0]])
```

np.eye()不仅能够创建对角线元素为 1 的 $n \times n$ 的二维数组（这种 $n \times n$ 的数组常常被形象地称为“方阵”，甚至有的资料直接称之为矩阵。而对角线都是 1，在数学上称为单位矩阵），还能够根据 k 值调整“对角线”位置——设置任何一个斜线方向为等效对角线（显然这是比喻的说法）。而 np.identity()所创建的就是一个不能调整“对角线”的单位矩阵（对角线元素是 1 的 $n \times n$ 的二维数组）。

```
In [52]: np.identity(4)
Out[52]: array([[ 1.,  0.,  0.,  0.],
                [ 0.,  1.,  0.,  0.],
                [ 0.,  0.,  1.,  0.],
                [ 0.,  0.,  0.,  1.]])
```

np.diag()比上述两个方法更灵活一些，从函数名称上看，也能知道这个函数是正宗操作对角线（Diagonal line）的函数，所以，读者在自己写程序的时候，也要注意命名的规范。让阅读代码的人能够望文生义，这才是最好的命名。

```
In [53]: np.diag([1, 2, 3, 4])
Out[53]: array([[1, 0, 0, 0],
                [0, 2, 0, 0],
                [0, 0, 3, 0],
                [0, 0, 0, 4]])
```

这个二维数组的对角线是由[1, 2, 3, 4]中的元素所指定的，并且 NumPy 自动根据对角线的个数确定数组各轴的元素数。与 np.eye()类似，np.diag()也可以调整对角线的位置。

```
In [54]: np.diag([1, 2, 3, 4], k=1)
Out[54]: array([[0, 1, 0, 0, 0],
                [0, 0, 2, 0, 0],
                [0, 0, 0, 3, 0],
                [0, 0, 0, 0, 4],
                [0, 0, 0, 0, 0]])
```

果然灵活。不仅如此，在下面的示例中还展现了其另一个作用。

```
In [55]: de = np.arange(16).reshape((4,4))
         de
Out[55]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])

In [56]: np.diag(de)
Out[56]: array([ 0,  5, 10, 15])

In [57]: np.diag(de, k=-1)
Out[57]: array([ 4,  9, 14])
```

In[55]创建了一个二维数组（np.arange()方法参考表 1-1-3，reshape()方法后续会讲解），在 In[56]中以该数组为参数传入 np.diag()中，得到了 de 数组的对角线元素组成的新数组。同理，在 In[57]中，调整了对角线位置，也得到由其元素组成的新数组。

（3）元素是等差和等比的数组

np.arange()是一个类似 Python 中的 range()的函数，通过它创建的是一维数组，数组的元素符合等差数列，即后一个元素值与前一个元素值的差等于指定的步长值。

```
In [58]: np.arange(1, 100, 3)
Out[58]: array([ 1,  4,  7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49,
                52, 55, 58, 61, 64, 67, 70, 73, 76, 79, 82, 85, 88, 91, 94, 97])
```

np.linspace()也可以得到由等差数列的数值组成的数组，但是，与上述 np.arange()有很大不同。

```
In [59]: np.linspace(1, 10, 4)
Out[59]: array([  1.,   4.,   7.,  10.])
```

In [59]规定该等差数列的开始值是 1，结束值是 10。但是，不同于 Python 中常规的“前包括，后不包括”的原则（参考 np.arange(10)的结果），这里的结束值也包括在数列之中（其实有一个参数 endpoint=True 来调控是否包括末尾的值，这个参数的默认值是 True，详情可以通过“np.linspace?”方法查看文档）。最后的 4，不是步长值，而是这个数列总共应该有的数值个数，或者说是数列的长度。

请参考函数的完整表述，理解上述说明：np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)。

np.logspace()在理解上较前面两个函数不那么直接了，所以先看示例，通过示例说明其作用。

```
In [60]: np.logspace(2, 3, num=4)
Out[60]: array([  100. ,  215.443469,  464.15888336, 1000. ])
```

先看 In [60]输入的参数，2 表示开始值，3 表示结束值，4 表示数量，即共有 4 个数。再看返回的结果，100 显然是 10^2，1000 则是 10^3，或者 2 是以 10 为底的 100 的对数，那么以 10 为底的 215.443469 的对数是多少？

```
In [61]: import math
         math.log10(215.443469)
         math.log10(464.15888336)
Out[61]: (2.333333333326906, 2.666666666665471)
```

看到上面的计算结果不难得知，Out[60]中的数组各个元素的以 10 为底的对数分别是[2, 7/3, 8/3, 3]，这个数列是以 1/3 为步长值均匀分布的，所以 np.logspace()返回的是在对数尺度上均匀间隔分布的数值。

如果从 np.logspace(2, 3, num=4)的参数进行理解，2 和 3 是两个对数结果，4 是对数个数，其完整的函数格式为：

```
np.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)
```

默认底是 10.0（base=10.0），可以根据需要修改。再看返回结果中的元素数字，分别是[$10^{6/3}$, $10^{7/3}$, $10^{8/3}$, $10^{9/3}$]，即得到的其实是一个等比数列。总结一句，np.logspace()返回的是由等比数列元素组成的数组，等比数列的开始值是 $base^{start}$，结束值是 $base^{end}$（视 endpoint=True 而定是否包含结束值）。

（3）创建自定义类型的数组

通过前面的学习，读者已经知道数组中的元素必然是某种类型，表 1-1-1 中的类型都是 NumPy 内置的类型。在 Python 中也有类似的现象，比如内置了字符串、列表、字典等对象类型。此外，在 Python 中我们还可以自定义对象类型（若对此尚有疑惑，请阅读《跟老齐学 Python：轻松入门》的有关章节）。那么，在 NumPy 中有没有给我们提供这样一种自定义元素类型的可能呢？

有！这是必须的。

因为这个社会是复杂的，即便是程序员，也要应对复杂的事务，仅有 NumPy 内置的几种类型是不能应付繁杂的现实业务需求的，必须允许我们自定义类型。

创建自定义类型，使用的方法是 np.dtype()。注意区分，这里所展示的不是数组对象的属性（data.dtype，data 是一个数组对象，其属性返回值是元素类型），而是 np（NumPy）的一个方法。

```
In [62]: my_type = np.dtype({"names":['book', 'version'], "formats":['S40', np.int]})
         my_type
Out[62]: dtype([('book', 'S40'), ('version', '<i8')])
```

my_type 就是自定义的 dtype 对象，其参数是一个字典，字典有两个键“names”和“formats”——键必须是这两个，通过两个键及其对应的值来定义类型的基本结构。这两个键的值分别是两个列表。“names”的值是['book', 'version']，规定了 my_type 的两个字段名称；“formats”的值是['S40', np.int]，依次规定了相应字段的类型，即 book 字段的类型是 S40，对照表 1-1-1 可知，此字段是长度为 40 字节的字符串类型，version 字段的类型是整数型。

然后创建一个新的数组，其元素类型为 my_type。

```
In [63]: my_books = np.array([("learn Python", 2), ('learn Django', 1)], dtype=my_type)
         my_books
Out[63]: array([(b'learn Python', 2), (b'learn Django', 1)],
               dtype=[('book', 'S40'), ('version', '<i8')])

In [64]: my_books.dtype
Out[64]: dtype([('book', 'S40'), ('version', '<i8')])
```

In[62]是一种定义类型的格式，下面的格式也可以采用。

```
In [65]: my_type2 = np.dtype([('book','S40'), ('version', np.int)])
```

通过自定义的元素类型 my_tpye 创建数组对象 my_books 后，可以用如下方式获得相应字段的值。

```
In [66]: my_books['book']
Out[66]: array([b'learn Python', b'learn Django'],
               dtype='|S40')

In [67]: my_books['book'][0]
Out[67]: b'learn Python'

In [68]: my_books[0]
Out[68]: (b'learn Python', 2)

In [69]: my_books[0]['book']
Out[69]: b'learn Python'
```

在 Out[64]和 Out[66]结果中，有“<”和“|”符号，它们表示的是字段值的字节顺序。“|”表示忽略字节顺序，“<”表示低位字节在前，“>”表示高位字节在前。

根据自定义的数据类型 my_type 得到的数组对象 my_books 颇似一个小型的数据库表，字段名分别是 book 和 version，并且一个是字符串类型，另一个是整数型。正因为有如此相似之处，我们就会想到，里面的具体记录是不是可以修改呢？

```
In [70]: my_books[0]['book'] = "learn Python with laoqi"
         my_books
Out[70]: array([(b'learn Python with laoqi', 2), (b'learn Django', 1)],
               dtype=[('book', 'S40'), ('version', '<i8')])
```

很有意思。能不能增加记录呢？

```
In [71]: my_books[2]['book'] = "Data Analysis"
---------------------------------------------------------------------------
IndexError                               Traceback (most recent call last)
<ipython-input-194-c2ebad39d92f> in <module>()
----> 1 my_books[2]['book'] = "Data Analysis"

IndexError: index 2 is out of bounds for axis 0 with size 2
```

这样增加内容的企图是不能得逞的，因为一个数组一旦确定，其轴的数量就不能变化了，所以跟数据库表还是有区别的。如果要让数组形状发生变化，必须用其他方法（后续讲解），但那些方法都是新建一个数组。

（4）用 from 系列函数创建数组

使用 dir(np)查看，会发现若干个以 from 开始的方法名称，使用这些方法也可以创建数组，它们分别是 frombuffer、fromfile、fromfunction、fromiter、fromregex、fromstring。下面就对这几个方法的使用做简单介绍。如果读者有兴趣，可以抛开下文讲述，使用类似“np.frombuffer?”的方法，逐个查看其帮助文档。下文的内容虽然是中文的，但是也不能替代文档，所以，即使阅读完以下内容，还是建议读者再次阅读文档。

当查看 frombuffer 文档时，会看到如下示例。

```
>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['w', 'o', 'r', 'l', 'd'],
      dtype='|S1')
```

这个示例说明了 np.frombuffer()的基本使用方法，但是，如果真的按照这个示例去调试，就会报错（当读者阅读本书的时候，或许使用的是更高版本的 NumPy，可能文档已经更新）。笔者是用下面的方式把上述示例调试通过的。

```
In [72]: np.frombuffer(b'hello world', dtype='S1', count=5, offset=6)
Out[72]: array([b'w', b'o', b'r', b'l', b'd'],
               dtype='|S1')
```

官方文档中的这个小问题，使笔者不禁想起古人说的“尽信书，则不如无书”（孟子），还有“纸上得来终觉浅，绝知此事要躬行”（陆游），所以，一定要动手敲代码，包括但不限于本书的代码，在阅读的时候一定要动手敲，方能“绝知此事”。

前面已经讲解了学习函数的方法，所以这里不再逐个介绍，只介绍 np.fromfunction()的使用方法。

文档中显示这个函数是 np.fromfunction(function, shape, **kwargs)，参数列表中的 function 是一个函数对象（在 Python 中函数是对象，对此的理解请阅读《跟老齐学 Python：轻松入门》有关章节），shape 是要创建的数组形状。

```
In [73]: def foo(x):
             return x + 1
         np.fromfunction(foo, (5,), dtype=np.int)
Out[73]: array([1, 2, 3, 4, 5])
```

fromfunction()中的参数“(5,)”表示数组的形状，即 0 轴有 5 个元素，这 5 个元素的索引依次是 0、1、2、3、4，这 5 个整数被逐个传给函数 foo 并在计算后返回结果，最终得到数组 array([1, 2, 3, 4, 5])。

```
In [74]: np.fromfunction(lambda i, j: (i+1)*(j+1), (9,9), dtype=np.int)
Out[74]: array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9],
                [ 2,  4,  6,  8, 10, 12, 14, 16, 18],
                [ 3,  6,  9, 12, 15, 18, 21, 24, 27],
                [ 4,  8, 12, 16, 20, 24, 28, 32, 36],
                [ 5, 10, 15, 20, 25, 30, 35, 40, 45],
                [ 6, 12, 18, 24, 30, 36, 42, 48, 54],
                [ 7, 14, 21, 28, 35, 42, 49, 56, 63],
                [ 8, 16, 24, 32, 40, 48, 56, 64, 72],
                [ 9, 18, 27, 36, 45, 54, 63, 72, 81]])
```

这里得到的是一个形状为(9, 9)的二维数组，其实就是九九乘法表的结果。对 In [74]的计算过程，可以用图 1-1-4 来理解。

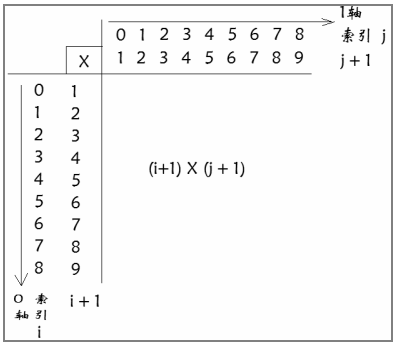


图 1-1-4　理解乘法表的运算

至此，我们已经对数组有了初步的认识。在实际的数据处理中，面对的可能是一个具有较大数据量的数组，也可能不是一维数组，二维数组，而是三维数组，甚至更高维度的数组。我们也许会关心它的某一部分数据，那么，如何从一个数组中取出一部分数据？取出来的这部分数据跟原来的数组有什么关系？且看下节讲述。

1.2　数组的索引和切片

在 Python 中，有一类数据被称为“序列”，比如字符串、列表等，这种类型的对象，每个元素都有索引。如果用“序列”的观点来看数组，数组中的每个元素是位置固定的，比如前面演示的“九九乘法表”二维数组，可以用类似坐标系中的坐标确定每个元素的位置，也就是说，可以用“索引”来表示数组中的每个元素。

本节将介绍在数组中运用索引，以及通过索引实现对数组的切片——在列表中，这些都曾经有过。

1．数组的轴

关于数组的轴，在前面曾经提过，这里再次单独介绍，是因为对它的正确理解事关重大，到底有多重大，请跟随本书一探究竟。

```
In [1]: import numpy as np
In [2]: a = np.arange(24).reshape((2, 3, 4))
        a
Out[2]: array([[[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]],

               [[12, 13, 14, 15],
                [16, 17, 18, 19],
                [20, 21, 22, 23]]])

In [3]: a.ndim, np.ndim(a)
Out[3]: (3, 3)
```

```
In [4]: a.shape
Out[4]: (2, 3, 4)
```

In[2]创建了一个数组，从 In[3]的结果可以知道，所创建的数组 a 的维度是 3，np.ndim(a)是另外一种获得数组维度数的方法（a.ndim 中的 a 是数组对象，所以这里的 ndim 是数组对象的属性；np 即 NumPy，是一个类，np.ndim()中的 ndim()是 np 类的方法）。In[4]得到了该数组的形状，描述其形状的元组(2, 3, 4)的长度就是数组的维度数，这个结论我们再次提到——凡是反复说的就是重要的。关于数组的维度和形状，请读者一定要区分开。

以前遇到过一维、二维的数组，In[2]中创建的是三维数组，不管是多少维度的数组，它的“轴”的命名是有规律的，为此画一张图（抽象派画法），一图胜千言，如图 1-2-1 所示。

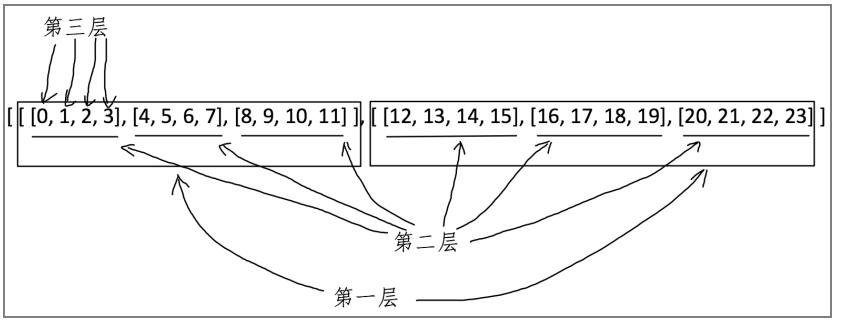


图 1-2-1　理解多维数组的轴

看了图，似乎更乱了。请读者深吸一口气，然后缓缓吐出，气定神闲，慢慢阅读下文，并对照图 1-2-1。

图 1-2-1 中把数组 a 的所有元素摆在一行。按照从外到内的顺序观察元素的层级，一共分为三层。第一层是两个大列表（分别记作 L0、L1）；第二层是 L0、L1 里面的各三个列表（L0 列表里面的第二层记作 L00、L01、L02，L1 列表里面的第二层记作 L10、L11、L12）；第三层是 L00、L01、L02 和 L10、L11、L12 列表中的整数类型元素。按照图 1-2-1 所示，要描述数组中“12”这个数字所在的位置，可以用下面的文字：数组 a 的第一层的第二个元素的下一层（第二层）的第一个元素的下一层（第三层）的第一个元素数字 12。

读完了感觉有点头晕。

NumPy 的创造者当然不像笔者这么傻，他们为数组创造了一个非常简单的方式，那就是为每层标记一个序号，官方说法叫做“索引”，通过索引说明某个元素就非常明确了，这种做法我们在 Python 的字符串、列表中都用过，并且在拙作《跟老齐学 Python：轻松入门》中还用梁山好汉做比喻给予了说明。

NumPy 的数组不都是一维的，从图 1-2-1 已经看出，如果有多个层级（多维），则单一的“索引”无法描述清楚是哪一层的元素，所以要同前面所言那样声明具体层级。NumPy 为了解决此问题，就规定：按照从外到内每一层是一个轴。因为是从 0 开始计数的，所以第一层就是（第）0 轴，第二层就是（第）1 轴，第三层就是（第）2 轴。每个轴上的元素，依次从 0 开始计数（该轴上的元素，即为该层级的单元），比如：

```
In [5]: a[1]
```

```
Out[5]: array([[12, 13, 14, 15],
               [16, 17, 18, 19],
               [20, 21, 22, 23]])

In [6]: a[1][0][0]
Out[6]: 12
```

In[5]的 a[1]表示对于数组 a 取其 0 轴方向索引是 1 的元素；In[6]的含义，用图画出来更明确，如图 1-2-2 所示。

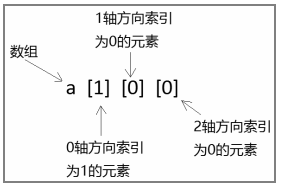


图 1-2-2　轴和索引

按照图 1-2-2 所描述的，依次取得相应索引所对应的元素，最后得到了数字“12”。可见，定义“数组的轴”为我们操作数组元素创造了非常便捷的方式。通过后面的讲述，读者可以进一步理解，通过数组的轴，我们可以任意操作指定方向上的元素。

非要把“轴”形象化，这是我们的大脑最喜欢的。运用自己的空间想象力，构建由轴组成的空间模型。一维就相当于一条线，二维就相当于一个面，三维呢？以 In[2]所建立的三维数组为例，可以想象一个 2 层的楼，第一层是 a[0]，第二层是 a[1]，然后每一层又分别是一个二维的面。如果是四维或者更高维度呢？这就超出我们的空间想象能力了，通常人能够直觉想象出来的只能到三维，维度再多的数组，就不得不依靠前面所说的方式从外到内依次定义轴来描述了（这个结论其实有些武断，总有聪明人可以把更高维度形象地描述出来，请阅读 http://www.itdiffer.com/article/33）。

2．根据索引取得数组的元素

因为 NumPy 是对数据进行操作的库，所以数学中的一个称呼“下标”也被引入了，其实“下标”和“索引”两个术语所说的是同一个对象，本书就不对二者进行区分了（如果非要区分，读者可以进行研究）。在下文中，读者会看到两个术语被交替使用。

（1）下标是整数

这个标题似乎是“废话”，索引（下标）本来就是整数呀，为什么非要用此标题呢？稍安勿躁。

```
In [7]: b = np.linspace(0, 100, 5)
        b
Out[7]: array([   0.,   25.,   50.,   75.,  100.])

In [8]: b.shape
Out[8]: (5,)

In [9]: b[3]
Out[9]: 75.0
```

像 b[3]那样，b 是一个数组，后面跟随方括号[ ]，写在方括号里面的就是所谓的“下标”。

数组 b 的维度是一维，b[3]的下标 3 就是整数，这跟列表很像，不用多解释了，下面再看二维数组的情况。

```
In [10]: c = np.logspace(1, 3, 12).reshape(3, 4)
         c
Out[10]: array([[   10.        ,    15.19911083,    23.101297  ,    35.11191734],
                [   53.36699231,    81.11308308,   123.28467394,   187.38174229],
                [  284.80358684,   432.87612811,   657.93322466,  1000.        ]])

In [11]: c.shape
Out[11]: (3, 4)

In [12]: c[1]
Out[12]: array([  53.36699231,   81.11308308,  123.28467394,  187.38174229])

In [13]: c[1][2]
Out[13]: 123.28467394420659
```

In[10]创建了一个由等比数列组成的二维数组，从 Out[10]结果可以看出，这个数组的 0 轴上有 3 个元素，1 轴上有 4 个元素。c[1]得到的是 0 轴上的第二个元素，即 Out[12]所示，显然它又是一个一维数组，用前面对一维数组的操作可知，c[1][2]得到一维数组 c[1]中索引是 2 的元素。

c[1][2]这种写法，还可以简化为：

```
In [14]: c[1, 2]    #或者 c[(1, 2)]
Out[14]: 123.28467394420659
```

在数组对象后面的[ ]中，按照从左到右的顺序，第一个整数对应的是 0 轴上的索引，第二个整数对应的是 1 轴上的索引，所得结果就是交点位置的数值——类似坐标点。

以上演示了一维数组和二维数组的情况，三维数组亦然，不再赘述。

当我们用这种方式得到元素或者数组后，对于原来的数组没有任何影响，而是新生成了一个对象。读者可以在上述操作之后再次查看数组 c，它并没有因为 In[12]等操作而少了一部分——这与 Python 中的序列对象雷同。

与列表一样，对于数组而言，每个索引对应的值可以通过索引（下标）进行修改。

```
In [15]: b
Out[15]: array([   0.,   25.,   50.,   75.,  100.])

In [16]: b[1] = 100
         b
Out[16]: array([   0.,  100.,   50.,   75.,  100.])

In [17]: c
Out[17]: array([[   10.        ,    15.19911083,    23.101297  ,    35.11191734],
                [   53.36699231,    81.11308308,   123.28467394,   187.38174229],
                [  284.80358684,   432.87612811,   657.93322466,  1000.        ]])

In [18]: c[1] = 1000
         c
```

```
Out[18]: array([[   10.       ,    15.19911083,    23.101297 ,    35.11191734],
                [ 1000.       ,  1000.       ,  1000.       ,  1000.       ],
                [  284.80358684,   432.87612811,   657.93322466,  1000.       ]])

In [19]: c[0, 2] = 99    #或者 c[0][2] = 99
         c
Out[19]: array([[   10.       ,    15.19911083,   99.       ,    35.11191734],
                [ 1000.       ,  1000.       ,  1000.       ,  1000.       ],
                [  284.80358684,   432.87612811,   657.93322466,  1000.       ]])
```

In[16]修改了数组中 b[1]的值，从 Out[16]可以看到修改结果，这也没有什么特别的地方，跟以往的列表一样。不过 In[18]的操作就不同了，c[1]是 0 轴的第二个元素，而这个元素就是一个列表，其中包含 4 个浮点数，c[1] = 1000 的结果是让这个列表中的所有元素都为 1000（最终结果要服从列表中对元素类型的规定），Out[18]显示了结果。

In[19]中的注释请读者关注，前面已经提到过，c[0, 2] == c[(0, 2)] == c[0][2]，这三种写法相当，都是最终锁定某一个最小单位的元素。

不仅是二维数组，多维数组亦然。图 1-2-3 显示了读取数组最小单位元素的方法，即在[ ]里面，按照从 0 轴开始的顺序依次写各轴上的索引，最终得到所要的元素。这个原则也适用于一维数组，因为它就一个 0 轴，所以只能写一个整数。

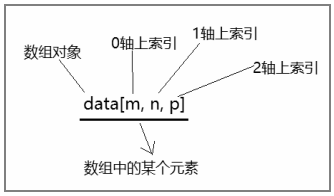


图 1-2-3　读取数组中某个元素

读者不妨按照图 1-2-3 所示的原则，尝试针对不同维度的数组，从中获取指定元素。

不管是一维数组还是多维数组，下面的情况总是会报错。

```
In [20]: b[5]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-38-ca5d543a082a> in <module>()
----> 1 b[5]

IndexError: index 5 is out of bounds for axis 0 with size 5

In [21]: c[3]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-39-91503357a59f> in <module>()
----> 1 c[3]

IndexError: index 3 is out of bounds for axis 0 with size 3
```

请认真阅读报错信息。在 Python 列表中也不允许进行类似上面的操作。

综观上述历程，我们完全可以根据列表中索引的经验来理解本标题下的数组问题。但貌似还没有回答前面的问题——为什么用“下标是整数”标题？

（2）下标是列表

整数作为下标能够理解，但把列表作为下标，着实有点“丈二和尚摸不着头脑”了——只要理解了下标可以是“列表”，就理解前面所说“下标是整数”的“废话”了。还是稍安勿躁，继续看操作示例。

```
In [22]: three = b[0], b[2], b[3]  #如果试图用 b[0, 2, 3]，肯定会报错，读者能理解其原因吗
         three
Out[22]: (0.0, 50.0, 75.0)
```

上述操作不难理解，从数组 b 中依次按照索引取出 3 个元素，这样的操作还可以写成下面的形式。

```
In [23]: three2 = b[[0, 2, 3]]
         three2
Out[23]: array([  0.,  50.,  75.])
```

请注意 b[[0, 2, 3]]，在[ ]里面放置的下标是一个列表[0, 2, 3]（注意与 b[0, 2, 3]和 b[(0, 2, 3)]不同），返回对象是 Out[23]所示的一个数组，数组的元素分别是 b[0]、b[2]、b[3]所对应的值。而 In[22]所实施的操作，虽然结果中的元素（如 Out[22]所示）内容与 Out[23]中的内容一样，但 Out[22]和 Out[23]是两个不同的对象。

注意，我们这里所用的下标是列表，不是元组。如果使用 b[(0, 2, 3)]，会怎样？请读者亲自尝试。

作为下标的列表，也可以仅有一个元素，返回的是一个新的数组。

```
In [24]: b[[2]]
Out[24]: array([ 50.])
```

凭什么说以列表作为下标得到的是新数组，表现在何处？

```
In [25]: three2[1] = 999
         three2
Out[25]: array([   0.,  999.,   75.])

In [26]: b
Out[26]: array([   0.,  100.,   50.,   75.,  100.])
```

In[25]对新数组 three2 中的一个元素值进行了修改，并没有影响原来数组 b 的值（如 Out[26]所示）。所以，以列表作为下标从原数组中得到一些元素并组成了新数组，这个新数组和原有数组彼此相互独立，从内存的角度说它们没有公用一个内存。

以上演示了对于一维数组，下标如果是一个列表，则可以列表中元素值为索引依次从数组中取值，并且生成一个新的数组。那么二维数组会怎样？

```
In [27]: c     #原来的一个数组，如果读者没有这个数组，可以模仿自建
Out[27]: array([[   10.       ,    15.19911083,   99.       ,    35.11191734],
                [ 1000.       ,  1000.       ,  1000.       ,  1000.       ],
                [  284.80358684,   432.87612811,   657.93322466,  1000.       ]])
```

```
In [28]: c[[0, 2]]
Out[28]: array([[   10.       ,    15.19911083,    99.       ,    35.11191734],
                [  284.80358684,   432.87612811,   657.93322466,  1000.       ]])
```

In[28]中的下标是列表[0, 2]，从返回的结果中可以推算，这个列表代表了 0 轴方向上的两个元素，再进一步看看。

```
In [29]: c[[0, 2], [1, 2]]
Out[29]: array([  15.19911083,  657.93322466])

In [30]: c[0, 1], c[2, 2]
Out[30]: (15.199110829529339, 657.93322465756819)
```

在 In[28]的基础上，又增加了[1, 2]这个列表，表示的是在 Out[28]结果的基础上，再从 1 轴方向分别取出索引是 1 和 2 的值，即最终得到 c[0, 1]和 c[2, 2]，并生成新的数组。

总结以列表为下标获得数组元素的方法，可以用图 1-2-4 说明。

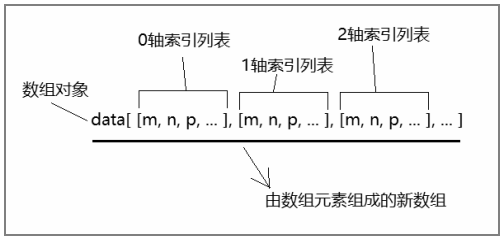


图 1-2-4　列表作为多维数组的索引

用列表作为下标，还可以通过重复索引多次得到有关元素。

```
In [31]: b[[2, 2, 2, 2]]
Out[31]: array([ 50.,  50.,  50.,  50.])
```

这其实是列表的问题，我们从来没有规定列表中的元素不可重复。

（3）下标是数组

数组的下标除可以是整数、列表外，还可以是数组，看来的确多样化。还是先看一维数组的情况。

```
In [32]: v = np.array([0, 1, 3])
         r = b[v]
         r
Out[32]: array([   0.,  100.,   75.])
```

In[32]中建立了数组 v，然后用它作为下标得到数组 b 中的某些元素，即 b[v]所得，Out[32]得到的结果依然是一个数组。

```
In [33]: b[np.array([2, 3, 2, 3, 2, 3])]
Out[33]: array([ 50.,  75.,  50.,  75.,  50.,  75.])
```

如果把列表看作“类数组”对象，那么“下标是数组”和“下标是列表”的操作就应该非常类似，所以对于二维数组，当下标是数组时，完全可以仿照下标是列表的情况理解。

```
In [34]: c[np.array([1,2])]
Out[34]: array([[ 1000.       ,  1000.       ,  1000.       ,  1000.       ],
```

```
               [  284.80358684,   432.87612811,   657.93322466,  1000.        ]])

In [35]: c[np.array([1, 2]), np.array([0, 1])]
Out[35]: array([ 1000.        ,   432.87612811])
```

第一个数组，获得的是 0 轴上的索引；第二个数组，对应的是 1 轴上的索引。当读者将上述结果和“下标是列表”的结果对照时，会发现非常相似。

下标是数组的情况，不仅局限在“下标是列表”那样的情况下。

```
In [36]: b
Out[36]: array([   0.,  100.,   50.,   75.,  100.])

In [37]: t = b == 50
         t
Out[37]: array([False, False,  True, False, False], dtype=bool)
```

In[37]是数组的一种操作，用“b == 50”这样一个简短的表达式，代替了一个 for 循环和一条 if 条件语句，其含义为：

```
for i in b:
    if i == 50:
        A.append(True)
    else:
        A.append(False)
```

上面的代码当然不能真的运行，旨在显示出 NumPy 的特点，“b == 50”直截了当地表明了上述诉求，并且返回一个数组，这个数组元素是布尔型的，然后用它作为下标。

```
In [38]: b[t]
Out[38]: array([ 50.])
```

要实现 In[38]的操作，有一个必需的前提，就是数组 b 和数组 t 的形状一样，两者的元素一一对应，按照对应关系，返回数组 t 中值为 True 的元素所对应的 b 中的元素，并组成数组对象。

```
In [39]: e = np.arange(10).reshape(2,5)
         t = e % 2 == 0
         e[t]
Out[39]: array([0, 2, 4, 6, 8])
```

对于二维或更高维度的数组，前面一一对应的原则依然适用。此外，对于多维的情况，还有一种特殊的机制。

```
In [40]: e
Out[40]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])

In [41]: t2 = np.array([False, True])
         e[t2]
Out[41]: array([[5, 6, 7, 8, 9]])
```

t2 没有跟 e 一一对应，但是将它作为下标依然有效，这里其实应用了一种被称为“广播”的机制，对此本书后续会详述。

以布尔型数组作为下标，其实是按照一定条件对原数组的元素进行筛选。

```
In [42]: e[e > 6]
Out[42]: array([7, 8, 9])
```

对于上述的操作，建议读者模仿前面的内容分步操作一番，从而理解其内涵。

以整数作为数组的下标，是一种“传统”的做法（此传统来自 Python 的列表），而以列表或者数组作为下标的方式，在以前学习 Python 时未见到，为此把这种方式称为“fancy indexing”，汉语常常翻译为“花式索引”，充满了“非传统”的味道。

3．数组的切片

根据下标，可以得到数组中的某个或者某几个元素，这是从数组中得到部分元素的一种方法。此外，还有另外一种被称为“切片”的方式，从这个名称和以前学习 Python 的经验可知，切片用来设定某个范围，根据这个范围从原数组中得到部分元素。

还是从简单的一维数组开始讲解。

```
In [43]: a = np.arange(10, 20)
         a
Out[43]: array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

In [44]: b = a[2: 8]
         b
Out[44]: array([12, 13, 14, 15, 16, 17])
```

In[44]的操作跟以往熟知的 Python 列表差不多，也遵守着“前包括、后不包括”的原则。

但是，要注意下面的新情况。

```
In [45]: b[0] = 12000
         b
Out[45]: array([12000,    13,    14,    15,    16,    17])

In [46]: a
Out[46]: array([10,    11,  12000,    13,    14,    15,    16,    17,    18,    19])
```

In[45]将新得到的数组 b 中的第一个元素的值进行了修改，结果见 Out[45]，同时原数组 a 中的元素也随之修改，见 Out[46]的结果。

之所以如此，是因为通过切片得到的数组和原数组共享了一个内存空间，通常称为公用同一个视图，当其中一个进行修改后，另一个也随之修改，这种特点与前面通过下标得到某些元素（尽管有的也生成数组）是不同的。

对于一维数组切片的其他方面，完全可以套用 Python 中列表的知识，比如下面所演示的。

```
In [47]: a[: : 2]
Out[47]: array([   10, 12000,    14,    16,    18])

In [48]: a[3:]
Out[48]: array([13, 14, 15, 16, 17, 18, 19])

In [49]: a[: : -1]
Out[49]: array([   19,    18,    17,    16,    15,    14,    13, 12000,    11,    10])

In [50]: a[: 6 : 3]
Out[50]: array([10, 13])
```

对于二维或者更多维度的数组进行切片，也是在一维数组的基础上进行的，只不过分别在每个轴方向上实施，最终获得重叠区域。为了能够使观察效果明显，先创建一个“别有用意”的二维数组。

```
In [51]: b = np.arange(0, 60, 10).reshape(-1, 1) + np.arange(0, 6)
         b
Out[51]: array([[ 0,  1,  2,  3,  4,  5],
                [10, 11, 12, 13, 14, 15],
                [20, 21, 22, 23, 24, 25],
                [30, 31, 32, 33, 34, 35],
                [40, 41, 42, 43, 44, 45],
                [50, 51, 52, 53, 54, 55]])
```

读者不用深究对 In[51]操作的理解，只要知道那样做能获得数组 b 即可，笔者会在后面讲解的。

下面就开始从刚刚创建的二维数组 b 上获得切片。毫无疑问，此数组有 0 轴和 1 轴两个方向，那么在每个轴上都可以看作是一维的（这是重要的分解思想，复杂事物都是由简单元素构成的，所以研究复杂事物的一个方法就是分解法）。

```
In [52]: b[1: 4]
Out[52]: array([[10, 11, 12, 13, 14, 15],
                [20, 21, 22, 23, 24, 25],
                [30, 31, 32, 33, 34, 35]])

In [53]: b[1 : 4, 2 : 5]
Out[53]: array([[12, 13, 14],
                [22, 23, 24],
                [32, 33, 34]])
```

b[1: 4]是在 0 轴方向上切片，得到了按照 0 轴方向上元素为单元的切片后的数组。

b[1 : 4, 2 : 5]是先在 0 轴方向上“切出一片”，然后在 1 轴方向上按照[2 : 5]的要求“切出”，最终得到 Out[53]所示的“那一片”，如图 1-2-5 所示。

b[1:4]

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

b[1:4, 2:5]

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

图 1-2-5　In[53]数组的切片

请读者认真观察图 1-2-5，体会切片的规则。

提醒读者注意，切片得到的数组跟原数组公用一个视图。这个很重要，请参考前述解释。

下面再用“代码+图示”的方式，展示一些常见的切片操作，读者可以根据下述示例，依次进行测试，并对照（依然以二维数组为例）结果。

```
In [54]: b[0]
Out[54]: array([0, 1, 2, 3, 4, 5])

In [55]: b[1, :]
```

```
Out[55]: array([10, 11, 12, 13, 14, 15])
```

In[54]和 In[55]数组的切片示意图如图 1-2-6 所示。

b[0]

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

b[1, :]

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

图 1-2-6　In[54]和 In[55]数组的切片

```
In [56]: b[: , 2]
Out[56]: array([ 2, 12, 22, 32, 42, 52])

In [57]: b[0 : 2, 0 : 2]
Out[57]: array([[ 0,  1],
                [10, 11]])
```

In[56]和 In[57]数组的切片示意图如图 1-2-7 所示。

b[:, 2]

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

b[0:2, 0:2]

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

图 1-2-7　In[56]和 In[57]数组的切片

```
In [58]: b[: : 2, : : 2]
Out[58]: array([[ 0,  2,  4],
                [20, 22, 24],
                [40, 42, 44]])

In [59]: b[: 3 , [0, 3]]
Out[59]: array([[ 0,  3],
                [10, 13],
                [20, 23]])
```

In[58]和 In[59]数组的切片示意图如图 1-2-8 所示。

b[::2, ::2]

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

b[:3, [0,3]]

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

图 1-2-8　In[58]和 In[59]数组的切片

从上述切片操作的示例中，不难总结出关于切片的两种理解方式。

理解一：先根据 0 轴的要求进行切片，然后将得到的结果根据 1 轴的要求再切片。

理解二：对原数组按照 0 轴和 1 轴两种要求进行切片，将两者的交集作为最终结果。

以上两种理解方式殊途同归。不过，对于更高维度的数组切片，按照“理解一”可能更容易一些。

现在我们已经能够通过下标或者切片获得数组中的一部分，这种针对数组的操作，是从轴和索引的角度完成的。此外，还有另外一种针对数组的操作，是通过属性和函数实现的，请看下节。

1.3 针对数组的操作

对于数组，除根据索引进行上节所述各种操作外，还有很多方法供我们使用，实现更加多样化的操作，例如本节将要介绍的变形、组合和分割等。

1．数组变形

所谓数组变形，就是将一个已有数组按照要求改变其形状后新生成一个数组，新数组的所有元素都来自于原数组，并且元素数量也保持不变，变的只有形状。

```
In [1]: import numpy as np
        np.arange(10)
Out[1]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [2]: np.arange(10).reshape((2, 5))
Out[2]: array([[0, 1, 2, 3, 4],
               [5, 6, 7, 8, 9]])
```

In[2]中使用的 reshape()是数组对象的方法。reshape()传入的参数就是新数组的形状。此外，np 也有这个方法，即 np.reshape()，先看一下这个方法的完整表述。

```
np.reshape(a, newshape, order='C')
```

其中，a 是原数组，newshape 是接收用元组表达的新数组的形状（一维数组可以是整数）。

在数组中，有很多数组对象的方法对应于 NumPy 类的方法，比如这里所说的 reshape()方法，可以是某个具体的数组对象的方法（表示为 a.reshape()），也可以是 NumPy 类的方法（表示为 np.reshape()）。

```
In [3]: a = np.arange(10)
        b = np.reshape(a, (2,5))
In [4]: b
Out[4]: array([[0, 1, 2, 3, 4],
               [5, 6, 7, 8, 9]])

In [5]: a
Out[5]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [6]: a.shape
Out[6]: (10,)

In [7]: b.shape
Out[7]: (2, 5)
```

a 是一个一维数组（In[3]），然后用 reshape()方法对数组 a 变形，得到新数组 b，b 相对 a

是一个新数组，两者不公用同一个视图，但是两个数组的元素数量和内容一样。

```
In [8]: c = np.reshape(b, (-1, 1))
        c
Out[8]: array([[0],
               [1],
               [2],
               [3],
               [4],
               [5],
               [6],
               [7],
               [8],
               [9]])
```

In[8]中提供的新数组形状有点奇怪，按照以前的经验，(–1, 1)不是数组形状，不可能存在–1 个元素。的确如此，在这个表示形状的数组中，1 轴上的数字没有错误，告诉 np.reshape()新数组 1 轴上有 1 个元素；但是 0 轴上，如果说有“–1”个元素，显然这样的“元素个数”是不存在的，这时候 NumPy 会自动推断 0 轴上应该有的元素个数，于是就得到了数组 c。当然，写其他不可能存在的数也可以，如下所示。

```
In [9]: c = np.reshape(b, (-2, 1))
```

In[8]和 In[9]的操作结果是一样的。

请读者注意比较下面两种操作中参数和结果的差异——两个结果貌似相同，实则有差异。

```
In [10]: np.reshape(b, (10,))
Out[10]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [11]: np.reshape(b, (1, 10))
Out[11]: array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

除使用 reshape()进行变形外，还可以使用数组的 shape 属性进行变形。

```
In [12]: b
Out[12]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])

In [13]: b.shape
Out[13]: (2, 5)

In [14]: b.shape = (1, 10)
         b
Out[14]: array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])

In [15]: b.shape = (10, 1)
         b
Out[15]: array([[0],
                [1],
                [2],
                [3],
                [4],
                [5],
```

```
               [6],
               [7],
               [8],
               [9]])
```

只是这样变形之后，改变了原有数组的形状，并不是新生成一个数组，所以要慎用。

对于数组对象，除用 np.ndarray.reshape()方法让数组变形外，还有一个方法可以将多维度的数组变成一个一维数组。

```
In [16]: p = b.flatten()
         p
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

顾名思义，np.ndarray.flatten()的作用就是将数组“扁平化”——变成一维数组，新的一维数组的内容完全复制了原数组内容，并且相对原数组不在同一个视图。

与 np.ndarray.flatten()等效的方法还有 np.reval()或 np.ndarray.ravel()，它们也能将数组转换为一维，只不过要注意它们的具体特点。

```
In [17]: ne = np.ravel(b)    #或者 b.ravel()
         ne
Out[17]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [18]: ne[1] = 111
         ne
Out[18]: array([ 0, 111,   2,   3,   4,   5,   6,   7,   8,   9])

In [19]: b
Out[19]: array([[  0],
                [111],
                [  2],
                [  3],
                [  4],
                [  5],
                [  6],
                [  7],
                [  8],
                [  9]])
```

认真观察上述操作，不难得出一个结论，np.ravel()或 np.ndarray.ravel()对数组进行变形后得到的新数组，与原数组公用同一个视图。

使用上述方法已经能够实现数组的各种变形了。但 NumPy 觉得还不够，还允许我们使用下标操作完成变形。

```
In [20]: data = np.arange(0, 5)
         c = data[:, np.newaxis]
         c
Out[20]: array([[0],
                [1],
                [2],
                [3],
                [4]])
```

In[20]的操作本质是什么？

请读者联想前面讲解的切片知识，data[:, np.newaxis]本质是对数组 data 进行切片，[ ]中逗号前面的部分，表示 0 轴方向上的所有元素，也就是 data 的所有元素；而逗号后面的部分说明了 1 轴方向上的元素在设置取值范围的时候，使用了 np.newaxis，这是什么？欲知晓，看文档。

```
In [21]: np.newaxis?
Type:       NoneType
String form: None
Docstring:   <no docstring>
```

原来 np.newaxis 就是 None，真的是这样吗？如果是这样，那么 In[20]中可以用 data[:, None]吗？有想法就要验证。

```
In [22]: data[:, None]
Out[22]: array([[0],
                [1],
                [2],
                [3],
                [4]])
```

实践证实了猜想，原来在做切片的时候，可以将某个轴设置为 np.newaxis 即 None，而这样做的结果实现了数组的一种变形。

再根据切片部分所习得的知识思考一个问题，通过上述方法变形得到的新数组相对于原数组，是在同一个视图吗？能不能检验你的结论？答案是“在同一个视图”，原因请参考前面关于切片的讲解。

In[20]把 data 数组由一维变成了二维，是将 1 轴设置为 np.newaxis。如果把 0 轴设置为 np.newaxis，会有什么效果呢？

```
In [23]: data.ndim
Out[23]: 1

In [24]: c.ndim
Out[24]: 2

In [25]: d = data[np.newaxis, :]
         d
Out[25]: array([[0, 1, 2, 3, 4]])

In [26]: d.ndim
Out[26]: 2

In [27]: data.shape, d.shape
Out[27]: ((5,), (1, 5))
```

针对这种在切片中使用 np.newaxis 实现变形的方法，NumPy 还提供了一个替代函数——np.expand_dims()。

```
In [28]: data2 = np.expand_dims(data, axis=0)
         data2
Out[28]: array([[0, 1, 2, 3, 4]])
```

```
In [29]: data2.shape
Out[29]: (1, 5)

In [30]: data3 = np.expand_dims(data, axis=1)
         data3
Out[30]: array([[0],
                [1],
                [2],
                [3],
                [4]])

In [31]: data3.shape
Out[31]: (5, 1)
```

np.expand_dims()的作用也是根据轴来调整数组形状。

使用数组对象或者 NumPy 的方法，以及用特殊的切片方式实现对数组的变形，这只是针对数组的一种操作，下面要说明的是数组的“分分合合、恩怨情仇”。

2．组合与分割

在 NumPy 中，实现数组与数组的组合，以及针对一个数组进行分割，有几个非常好用的方法。这些方法不仅丰富了针对数组的操作方式，更为解决复杂的实际问题提供了多种可选工具。这就好比行走江湖，不能只会“神行百变”，韦小宝擅长的还有各种其他功夫，在面对不同的实际场景时才能随机应变。

（1）水平组合

实现水平组合功能的函数形式是 np.hstack(tup)，其中参数 tup 是一个元组，包含即将被组合在一起的几个数组。下面的示例假设是二维数组，要求其 0 轴方向的形状一样，而 1 轴方向的形状可以不同。

```
In [32]: a = np.arange(9).reshape(3, 3)
         b = np.arange(12).reshape(3, 4)
         c = np.arange(15).reshape(3, 5)

In [33]: np.hstack((a, b))
Out[33]: array([[ 0,  1,  2,  0,  1,  2,  3],
                [ 3,  4,  5,  4,  5,  6,  7],
                [ 6,  7,  8,  8,  9, 10, 11]])

In [34]: np.hstack((a, b, c))
Out[34]: array([[ 0,  1,  2,  0,  1,  2,  3,  0,  1,  2,  3,  4],
                [ 3,  4,  5,  4,  5,  6,  7,  5,  6,  7,  8,  9],
                [ 6,  7,  8,  8,  9, 10, 11, 10, 11, 12, 13, 14]])
```

此外，还有实现水平组合的另外两种方法，一个是 np.stack()，另一个是 np.concatenate()，这两个函数的使用方法雷同，能够按照任何方向实现数组的组合。但是，一切都要尝试才能获得结果。

```
In [35]: np.stack((a, b), axis=1)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

```
<ipython-input-98-aa3fe05e7512> in <module>()
----> 1 np.stack((a,b), axis=1)

/usr/local/lib/python3.5/dist-packages/numpy/core/shape_base.py in stack(arrays, axis)
    352     shapes = set(arr.shape for arr in arrays)
    353     if len(shapes) != 1:
--> 354         raise ValueError('all input arrays must have the same shape')
    355
    356     result_ndim = arrays[0].ndim + 1

ValueError: all input arrays must have the same shape

In [36]: np.concatenate((a, b), axis=1)
Out[36]: array([[ 0,  1,  2,  0,  1,  2,  3],
                [ 3,  4,  5,  4,  5,  6,  7],
                [ 6,  7,  8,  8,  9, 10, 11]])
```

原来，np.stack()所要求的数组必须“Each array must have the same shape”（引自文档），而 np.concatenate()对被组合的数组的要求是“The arrays must have the same shape, except in the dimension corresponding to `axis` (the first, by default)”，所以，np.stack()必须按下面这样操作。

```
In [37]: m = a * 3
         a.shape == m.shape
Out[37]: True

In [38]: np.stack((a, m), axis=1)
Out[38]: array([[[ 0,  1,  2],
                 [ 0,  3,  6]],

                [[ 3,  4,  5],
                 [ 9, 12, 15]],

                [[ 6,  7,  8],
                 [18, 21, 24]]])
```

从组合的效果来看，np.stack()也迥异于 np.concatenate()。只有 In[36]的 np.concatenate((a,b), axis=1)与 np.hstack((a,b))效果等同。

（2）垂直组合

前面的水平组合是沿着 1 轴组合（对于二维数组而言如此，对于多维数组也是如此），那么这里的垂直组合，则是沿着与 1 轴垂直的方向（0 轴）组合。之所以用二维数组举例，是因为这种类型的数组较常用，更多维的数组不如二维数组使用广泛。

实现垂直组合的专有函数是 np.vstack()，有了前述经验，操作此方法应该简单了。

```
In [39]: np.vstack((a, b))
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-118-08bbbce58d8b> in <module>()
----> 1 np.vstack((a,b))

/usr/local/lib/python3.5/dist-packages/numpy/core/shape_base.py in vstack(tup)
```

```
    235
    236     """
--> 237     return _nx.concatenate([atleast_2d(_m) for _m in tup], 0)
    238
    239 def hstack(tup):

ValueError: all the input array dimensions except for the concatenation axis must match
exactly
```

错误再次出现，注意看提示信息。不要忘记 a、b 两个数组的形状。

```
In [40]: a.shape, b.shape
Out[40]: ((3, 3), (3, 4))
```

np.vstack()要求被组合的数组在 1 轴方向上的形状相同。

```
In [41]: b2 = b.T
         b2.shape
Out[42]: (4, 3)

In [43]: np.vstack((a, b2))
Out[43]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 0,  4,  8],
                [ 1,  5,  9],
                [ 2,  6, 10],
                [ 3,  7, 11]])
```

b.T 是对原数组进行转置，即将 0 轴和 1 轴互换，得到的新数组在 1 轴方向上与数组 a 形状相同，之后顺利实现了 In[43]的操作。

np.concatenate()在传入轴的参数之后，也能实现垂直组合，与上述操作等效。

```
In [44]: np.concatenate((a, b2), axis=0)
Out[44]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 0,  4,  8],
                [ 1,  5,  9],
                [ 2,  6, 10],
                [ 3,  7, 11]])
```

（3）其他组合

在组合的方式上，除常用的“水平组合”和“垂直组合”外，还有“深度组合”“行组合”“列组合”。因为这些组合的操作模式与前述类似，所以这里仅以举例说明，读者可以通过查看帮助文档深入学习有关内容。

```
In [45]: a    #承接前面创建的数组 a
Out[45]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])

In [46]: b = a * 3    #数组 b 与数组 a 的形状相同
```

```
        np.dstack((a, b))    #深度组合
Out[46]: array([[[ 0,  0],
                 [ 1,  3],
                 [ 2,  6]],

                [[ 3,  9],
                 [ 4, 12],
                 [ 5, 15]],

                [[ 6, 18],
                 [ 7, 21],
                 [ 8, 24]]])
```

为了形象地理解深度组合，可以把 a、b 分别想象成两个平面，一上一下，然后沿着竖直方向组合上下对应的元素，形成三个竖直的面，它们就是新生成的数组。

```
In [47]: one = np.arange(5)
         two = np.arange(5, 10)
         np.column_stack((one, two))
Out[47]: array([[0, 5],
                [1, 6],
                [2, 7],
                [3, 8],
                [4, 9]])
```

In[47]实现的是对两个一维数组的列组合。

如果用 np.column_stack()对二维数组实施列组合，其效果等效于 np.hstack()，读者可以自己尝试。

```
In [48]: np.row_stack((one, two))
Out[48]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])
```

np.row_stack()在 In[48]中实现的是对两个一维数组的行组合，如果操作的是二维数组，则效果等效于 np.vstack()所实现的垂直组合。

与组合相对应的操作就是分割。

（4）数组的分割

np.split()是一个比较通用的分割方法，其函数形式是 np.split(ary, indices_or_sections, axis=0)。看例子，明道理。

```
In [49]: a = np.arange(24).reshape(4, 6)
         a
Out[49]: array([[ 0,  1,  2,  3,  4,  5],
                [ 6,  7,  8,  9, 10, 11],
                [12, 13, 14, 15, 16, 17],
                [18, 19, 20, 21, 22, 23]])

In [50]: np.split(a, 2, axis=1)
Out[50]: [array([[ 0,  1,  2],
                 [ 6,  7,  8],
                 [12, 13, 14],
```

```
                [18, 19, 20]]), array([[ 3,  4,  5],
                [ 9, 10, 11],
                [15, 16, 17],
                [21, 22, 23]])]

In [51]: np.split(a, 2, axis=0)
Out[51]: [array([[ 0,  1,  2,  3,  4,  5],
                [ 6,  7,  8,  9, 10, 11]]), array([[12, 13, 14, 15, 16, 17],
                [18, 19, 20, 21, 22, 23]])]
```

np.split()方法根据 axis 来确定分割的方向。此外，针对每个方向的分割，也有专门的函数，就像组合那样。

```
In [52]: np.hsplit(a, 2)
Out[52]: [array([[ 0,  1,  2],
                [ 6,  7,  8],
                [12, 13, 14],
                [18, 19, 20]]), array([[ 3,  4,  5],
                [ 9, 10, 11],
                [15, 16, 17],
                [21, 22, 23]])]

In [53]: np.vsplit(a, 2)
Out[53]: [array([[ 0,  1,  2,  3,  4,  5],
                [ 6,  7,  8,  9, 10, 11]]), array([[12, 13, 14, 15, 16, 17],
                [18, 19, 20, 21, 22, 23]])]
```

除能够对二维数组进行分割外，还能对更高维度的数组进行分割，读者不妨自己测试一下，此处不再赘述。

对数组的操作还可以再深入一步——增、删、改元素。

3．改编元素

此处用“改编”，而不是“改变”，是有意为之。

所谓“改编”数组元素，意思是不仅能够修改数组中的已有元素，还能够进行增加、删除等操作。“修改”数组元素，在上一节关于索引的内容中已经介绍了，不足为奇。倒是增加、删除元素，有点奇怪了。因为一个数组一旦被创建，其形状确定，怎么能删除和增加元素呢？难道一个 3×4 的数组（12 个元素）可以随便少一个元素吗？非也。且看下文慢慢分晓。

首先看 np.append()方法，看到这个方法不由得想起列表中的 append()方法，对于列表对象而言，append()是向其追加元素，因为我们知道列表是可变的，那么数组呢？

```
In [54]: a = np.array([[1, 2, 3], [4, 5, 6]])
         b = np.array([[7, 8, 9]])
         a.ndim, b.ndim
Out[54]: (2, 2)

In [55]: r = np.append(a, b, axis=0)
         r
Out[55]: array([[1, 2, 3],
                [4, 5, 6],
```

```
                [7, 8, 9]])
In [56]: a
Out[56]: array([[1, 2, 3],
                [4, 5, 6]])

In [57]: b
Out[57]: array([[7, 8, 9]])
```

这里所创建的两个数组的维度是一样的，在 np.appen(a, b, axis=0)中，将数组 b“追加”到数组 a 中，并且是在 0 轴方向，结果是得到了一个新数组 r，数组 b 的元素被“追加”到了数组 a 在 0 轴方向的尾部。虽然也是 append()，但没有修改原有数组 a——与列表不同，是用数组 a 和数组 b 的结构和元素，重新建立了一个数组 r。所以此处的“追加”跟列表中的“追加”是有不同含义的，虽然外表有些相似。

```
In [58]: np.append(a, b)
Out[58]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

如果不声明轴，则意味着只是用原来两个数组的元素创建一个新的一维数组，如 Out[58]所示。

除“追加”外，还能“插入”，即某些元素“插入”到指定位置。

```
In [59]: a1 = a.flatten()
         a1
Out[59]: array([1, 2, 3, 4, 5, 6])

In [60]: r = np.insert(a1, 1, 99)
         r
Out[60]: array([ 1, 99,  2,  3,  4,  5,  6])

In [61]: a1
Out[61]: array([1, 2, 3, 4, 5, 6])
```

观察 In[60]的操作，将 99 插入到指定位置，并没有修改原数组，而是重新创建了一个数组，仿照 np.append()说法就是利用 a1 数组的元素和形状，以及要插入的元素 99 新建了一个数组。

如果仿照列表的操作，再有一个“删除”功能就最好了。NumPy 果然不负众望，的确有这个方法。

```
In [62]: a
Out[62]: array([[1, 2, 3],
                [4, 5, 6]])

In [63]: np.delete(a, 1, axis=0)
Out[63]: array([[1, 2, 3]])

In [64]: a
Out[64]: array([[1, 2, 3],
                [4, 5, 6]])
```

从上面的简要操作中已经看出，这个所谓的“删除”，事实上是重新创建了一个不含有指定元素的数组，对原来的数组丝毫没有影响。

以上简单介绍了 np.append()、np.insert()、np.delete()三个方法，说这三个方法能够“改编”

数组，其实都是创建一个新数组。特别建议读者认真阅读这三个方法的文档，不仅能够从更深层次理解这三个方法，而且掌握了一种学习方法。

在数据处理中，用数组来参与一些运算是必不可少的，也正是在数学运算中，NumPy 的优势才体现得淋漓尽致。

1.4 运算和通用函数

NumPy 的优势在于科学计算和数值处理。比如有一个比较大的列表，由很多整数组成，如果要对这些整数进行某种运算，在 Python 中不得不使用循环——特殊函数可以在一定程度上避免写循环语句，而使用 NumPy 则不需要写循环语句，用更直接的运算符号或者函数就能完成对众多数值的操作。本节所要介绍的算数运算、比较运算、逻辑运算和通用函数，都是元素级的操作，即针对数组的每个元素进行一对一的操作。

1．算数运算

我们已经熟知 Python 中的算数运算，比如这样一个题目：有一个列表[1, 20, 28, 37, 18, 56]，要求将列表中的每个元素乘以 3。在 Python 中，我们可以这么解决：

```
In [1]: lst = [1, 20, 28, 37, 18, 56]
        [i*3 for i in lst]
Out[1]: [3, 60, 84, 111, 54, 168]
```

完成上述题目，在 Python 中需要使用 for 循环（如 In[1]所示）。如果列表中的数据量增加（比如是所谓的“大数据”那么大。到底多大？说不清，反正很大——所以本书开篇说过了，“大数据”仅仅是商业上夺人眼球的宣传用语罢了），那么循环所耗费的时间也会增加，要想让程序“更快、更强”，就得使用 NumPy。

```
In [2]: import numpy as np
        a = np.array(lst)    #①
        a * 3
Out[2]: array([  3,  60,  84, 111,  54, 168])
```

In[2]中的①将列表转换为数组之后，直接执行“a * 3”，就实现了数组中每个元素跟 3 相乘，这里没有显式地使用循环就实现了 In[1]循环的效果，这就是 NumPy 的数组优势所在。

在“a * 3”这个表达式中，我们常称 3 为标量，称数组 a 为矢量或者向量。

通过上面的例子，我们已经看到了用数组进行算数运算的特点了，下面再按部就班地完整讲述。

```
In [3]: a = np.arange(10).reshape((2, 5))
        a
Out[3]: array([[0, 1, 2, 3, 4],
               [5, 6, 7, 8, 9]])

In [4]: b = a + 10
        b
Out[4]: array([[10, 11, 12, 13, 14],
               [15, 16, 17, 18, 19]])
```

```
In [5]: a - 10
Out[5]: array([[-10,  -9,  -8,  -7,  -6],
               [ -5,  -4,  -3,  -2,  -1]])

In [6]: a / 10
Out[6]: array([[ 0. ,  0.1,  0.2,  0.3,  0.4],
               [ 0.5,  0.6,  0.7,  0.8,  0.9]])
```

一个数组和一个标量进行加、减、乘、除运算，结果是数组中的每个元素都与该标量进行相应运算，并返回一个新数组。

那么，两个数组之间是否可以运算呢？

原则上讲，只有形状一样的数组之间才能进行运算，例如下面的计算示例中，a 和 b 两个数组的形状一样，它们之间进行算数运算时，就是对应位置的元素进行相应运算。

```
In [7]: a.shape == b.shape
Out[7]: True

In [8]: a + b
Out[8]: array([[10, 12, 14, 16, 18],
               [20, 22, 24, 26, 28]])

In [9]: a - b
Out[9]: array([[-10, -10, -10, -10, -10],
               [-10, -10, -10, -10, -10]])

In [10]: a * b
Out[10]: array([[  0,  11,  24,  39,  56],
                [ 75,  96, 119, 144, 171]])

In [11]: a / b
Out[11]: array([[ 0.        ,  0.09090909,  0.16666667,  0.23076923,  0.28571429],
                [ 0.33333333,  0.375     ,  0.41176471,  0.44444444,  0.47368421]])
```

如果两个数组的形状不同，则 NumPy 不支持完成上述算数运算，但是，某种情况除外，“原则性和灵活性”在 NumPy 中得到了统一。

所谓“某种情况”，就是 NumPy 能够将某个数组通过“广播”的方式进行临时转换，使得两个数组的形状符合“原则上讲的要求”。

```
In [12]: a
Out[12]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])

In [13]: m = np.arange(5)
         m
Out[13]: array([0, 1, 2, 3, 4])

In [14]: a + m
Out[14]: array([[ 0,  2,  4,  6,  8],
                [ 5,  7,  9, 11, 13]])
```

```
In [15]: n = np.arange(1, 3).reshape((-1, 1))
         a + n
Out[15]: array([[ 1,  2,  3,  4,  5],
                [ 7,  8,  9, 10, 11]])
```

数组 a 和数组 m 的形状不同（见 Out[12]和 Out[13]），但是两个数组的 1 轴长度一样，于是在相加的时候，NumPy 会自动将数组 m 进行广播（如图 1-4-1 所示），使数组 m 与数组 a 的形状一样后再相加。同理，数组 n 也是如此，只不过数组 n 是在 0 轴上与数组 a 相同。

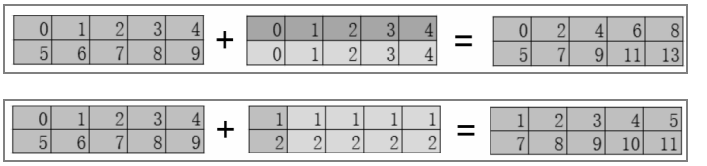


图 1-4-1　数组之间的运算

以上仅以加法为例说明了广播的含义，读者可以自行测试其他运算中的广播机制——注意，“广播”也不是无条件的，如上所述。

2．比较运算和逻辑运算

Python 中的比较运算符和逻辑运算符，在 NumPy 中依然适用。

```
In [16]: np.array([6, 4, 3]) > np.array([9, 5, 1])
Out[16]: array([False, False,  True], dtype=bool)
```

从返回的结果中不难看出，数组的比较运算是一一对应地进行比较，然后以数组形式返回比较结果。

```
In [17]: np.array([6, 4, 3]) > np.array([9, 5])
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-368-27e535beb1e5> in <module>()
----> 1 np.array([6, 4, 3]) > np.array([9, 5])

ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

比较运算同样要求两个数组的形状一样。

Python 中有 and、or 和 not 三个逻辑运算符，因为 NumPy 是第三方库，所以它不能把自己的逻辑运算也用这三个单词命名，而是使用 logical_and、logical_or、logical_not、logical_xor 等。

```
In [18]: np.logical_or(np.array([False, True]), np.array([True, False]))
Out[18]: array([ True,  True], dtype=bool)
```

首先回忆 Python 中“or”运算的过程（如果忘记，请参阅《跟老齐学 Python：轻松入门》有关章节内容），然后将回忆起来的知识运用到 In[18]，最终得到针对每个元素一对一“or”运算的数组（Out[18]的结果）。只是要注意，In[18]中没有使用“or”运算符，而是使用了 logical_or()函数。

在 Python 中，我们可以进行下面的运算。

```
In [19]: x = 3; y = 5
         x < y and x > y
```

```
Out[19]: False
```

然而，类似的操作在 NumPy 中，就有问题了。

```
In [20]: a = np.array([3,6,9])
         b = np.array([4, 5, 8])
         a < b and a > b     #①
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-382-9030be40eafc> in <module>()
----> 1 a < b and a > b

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any()
or a.all()
```

系统报错了。

千万不要看到报错信息就六神无主了，仔细阅读报错信息，是程序员必备的修养。

NumPy 能够推断我们的心思，看 ValueError 后面的信息，给我们提示了。要实现 In[20]中①的结果，可以使用 a.any()或者 a.all()，这是什么？请者顺着这个信息去查看帮助文档，就能理解这两个函数的含义，笔者在这里用示例说明。

```
In [21]: np.any(a<b) and np.any(a>b)
Out[21]: True

In [22]: np.all(a<b)
Out[22]: False

In [23]: (a<b).all()
Out[23]: False
```

- np.any(a, axis=None, out=None)，a 是类数组对象，只要 a 中（或者某个指定的轴中）有一个元素是 True，则返回 True，否则返回 False。
- np.all(a, axis=None, out=None)，如果类数组对象 a（或者某个指定的轴中）所有元素都是 True，则返回 True，否则返回 False。

In[20]代码的意图，应该用 In[22]或者 In[23]来实现。

3．通用函数

NumPy 中有一类函数被称为“通用函数”（英文是 Universal Function，简写为 UFunc），能对数组中的每个元素进行操作，即元素级的函数，并且这些函数都是在 C 语言级别实现的，因此它们的计算速度非常快。本节主要介绍部分与运算相关的通用函数。

（1）内置函数

NumPy 有一些默认的通用函数，按照参数的个数可将它们进行分类，一个参数的称为一元函数，两个参数的称为二元函数，表 1-4-1 中列出的是算数运算中常用的一元函数。

下面用示例来说明一元函数的使用方法，其他没有涉及到的函数，请读者自己尝试。

```
In [24]: alpha = np.linspace(-1, 1, 11)
         y = np.sin(np.pi * alpha)
         np.round(y, decimals=3)
```

```
Out[24]: array([-0.   , -0.588, -0.951, -0.951, -0.588,  0.   ,  0.588,  0.951,
                0.951,  0.588,  0.   ])
```

表 1-4-1　常用的一元函数

函　数	说　明
np.sin，np.cos，np.tan	三角函数
np.arcsin，np.arccos，np.arctan	反三角函数
np.sinh，np.cosh，np.tanh	双曲三角函数
np.arcsinh，np.arccosh，np.arctanh	反双曲三角函数
np.sqrt	求平方根
np.exp	计算自然指数
np.log，np.log2，np.log10	计算对数（依次以 e、2、10 为底）

除一元函数外，还有二元函数，看下面的示例。

```
In [25]: a, b       #承接前面操作创建的数组 a 和数组 b
Out[25]: (array([3, 6, 9]), array([4, 5, 8]))

In [26]: np.add(a, b)
Out[26]: array([ 7, 11, 17])
```

In[26]显示了 np.add()的应用，从效果上看与使用运算符“+”号的结果是一样的。当然，在实际的项目中，读者可以任意选择使用，不过使用运算符更方便、直观。

NumPy 中还有其他二元函数，表 1-4-2 中只列出了一部分。

表 1-4-2　部分二元函数

函　数	说　明
np.add，np.subtract，np.multiply，np.divide	算数运算函数
np.equal，np.not_equal，np.less，np.less_equal，np.greater，np.greater_equal	比较运算函数
np.power	指数运算
np.remainder	得到余数
np.reciprocal	返回倒数（不要使用整数型）
np.real，np.imag，np.conj	返回复数的实部、虚部和完整的复数
np.sign，np.abs	得到对象符号和绝对值
np.floor，np.ceil，np.rint	结果都是取整
np.round	四舍五入

在三角函数中，$\sin^2(x)+\cos^2(x)=1$ 是大家都知道的，如果使用二元函数，怎么实现呢？

```
In [27]: np.add(np.power(np.sin(alpha), 2), np.power(np.cos(alpha), 2))
Out[27]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])

In [28]: np.sin(alpha)**2 + np.cos(alpha)**2
Out[28]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

In[27]和 In[28]效果相当，只是前者使用了二元函数。

如果读者查看了每个函数的帮助文档，会发现有一个参数 out，这个参数的含义是什么？

```
In [29]: a = np.arange(10.0, 100.0, 2)
         b = np.logspace(2, 4, a.shape[0])
         m = np.empty_like(a)    #①
         a.dtype, b.dtype, m.dtype
Out[29]: (dtype('float64'), dtype('float64'), dtype('float64'))

In [30]: np.remainder(b, a, out=m)
Out[30]: array([  0.       ,   3.03363182,  11.28467394,   8.88745095,
                  7.9911083 ,   8.76124758,  11.38174229,  16.05675382,
                 23.01297001,   4.50209057,  14.80358684,  28.22776602,
                 11.11917342,  29.86037025,  14.87612811,   0.63808631,
                 29.66992312,  20.55309755,  13.93322466,  10.52715427,
                 11.13083079,  16.62802021,  28.       ,   46.33631817,
                 14.84673944,  48.87450954,  31.91108295,  23.61247579,
                 25.81742286,  40.56753822,   0.12970008,  45.02090568,
                 36.03586844,  46.27766017,   1.19173422,  58.60370255,
                 64.76128108,  18.38086306,   4.69923121,  29.53097555,
                  9.33224658,  37.27154266,  27.3083079 ,  78.28020211,   4.       ])
In [31]: m
Out[31]: array([  0.       ,   3.03363182,  11.28467394,   8.88745095,
                  7.9911083 ,   8.76124758,  11.38174229,  16.05675382,
                 23.01297001,   4.50209057,  14.80358684,  28.22776602,
                 11.11917342,  29.86037025,  14.87612811,   0.63808631,
                 29.66992312,  20.55309755,  13.93322466,  10.52715427,
                 11.13083079,  16.62802021,  28.       ,   46.33631817,
                 14.84673944,  48.87450954,  31.91108295,  23.61247579,
                 25.81742286,  40.56753822,   0.12970008,  45.02090568,
                 36.03586844,  46.27766017,   1.19173422,  58.60370255,
                 64.76128108,  18.38086306,   4.69923121,  29.53097555,
                  9.33224658,  37.27154266,  27.3083079 ,  78.28020211,   4.       ])
```

请注意 In[29]的①创建了一个空数组，试图用这个数组来容纳 b 除以 a 的余数。

若要让 In[30]操作成功，必须保证数组 m 和前面两个数组（或者广播之后的数组）类型和形状一样。In[30]中的 out=m，即声明该计算结果保存到数组 m 中。

如果仅用内置的通用函数还不足以满足需要，则可以使用自定义通用函数——就如同 Python 中的对象，除使用“内置”的对象外，还能“自定义”。

（2）自定义通用函数

写一个关于 BMI（身体质量指数，简称体质指数，英文为 Body Mass Index，简称 BMI）的函数，通过这个函数判断一个人的胖瘦。

```
In [32]: def bmi(height, weight):
             bmi_index = weight / height ** 2
             if bmi_index >18 and bmi_index < 25:
                 return 0
             elif bmi_index <= 18:
                 return -1
             else:
```

```
            return 1
        bmi(1.55, 45)    #①
Out[32]: 0
```

在这个 bmi()函数中，比较粗略地对胖瘦进行了三档划分，即正常返回 0，偏瘦返回–1，超重返回 1。向此函数传入了某退役老师的身高体重（数据来自网络），返回值是 0。

用 In[32]中的①，一次只能判断一个人的 BMI，如果要判断很多人的 BMI，怎么处理？循环吗？这是一种方法，但在数组中，还有另外一种方法。

```
In [33]: h  = (2.2 - 1.4)* np.random.random_sample((10,)) + 1.4
         h = np.round(h, 2)
         w = (96 - 30) * np.random.random_sample((10,)) + 30
         w = np.round(w, 2)
```

要检验 bmi()函数对数组是否成立，需要有表示身高和体重的数组。按照一般想法，可以找若干个人，分别测量他们的身高和体重并记录数据。这是一种方法，但笔者没有能力找那么多人，怎么办？我们可以编造数据——不是“瞎编”，编得也要合情合理。

In[33]中使用 np.random.random_sample()方法（请读者用笔者一贯提倡的方法自行查看 np.random 的文档说明）生成 10 个随机数，同时还要约束这些数字在某个范围内（身高不是任意范围的数字），这里我们选择了[1.4, 2.2]这个范围的任意 10 个数字作为身高测试数据（包括 1.4，不包括 2.2，单位是 m），并且取小数点后两位小数。用同样的方法，也获得体重（单位 kg）的 10 个随机数组成的数组 w。

下面就开始“见证奇迹”了。

```
In [34]: bmi(h, w)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-31-36d911c3b98f> in <module>()
----> 1 bmi(h, w)

<ipython-input-1-f0c6c75a5af8> in bmi(height, weight)
     1 def bmi(height, weight):
     2     bmi_index = weight / height ** 2
----> 3     if bmi_index >18 and bmi_index < 25:
     4         return 0
     5     elif bmi_index <= 18:

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

“演”砸了！

别灰心，耐心、认真地阅读报错信息。

报错信息中说函数 bmi()的第三行代码有问题。从代码的角度看，使用 np.any()或者 np.all()修改这一行是可以的，但如果真的这样做了，就失去了我们本来的目标——自定义通用函数。

```
In [35]: np.array([bmi(h[i], w[i]) for i in range(len(h))])
Out[35]: array([ 0,  1,  0, -1,  0, -1,  0, -1,  0, -1])
```

还是用功能强大的列表解析完成吧。

然而，这不是我们的终极目标，真正的目标是下面这个示例。

```
In [36]: bmi_ufunc = np.frompyfunc(bmi, 2, 1)
         bmi_ufunc(h, w)
Out[36]: array([0, 1, 0, -1, 0, -1, 0, -1, 0, -1], dtype=object)
```

比较 Out[35]和 Out[36]的结果，一样！说明使用 In[36]操作同样实现了 In[35]的要求。

为什么函数 bmi()经过 np.frompyfunc()处理之后，就成为能够对数组元素进行操作的函数了呢？

根源在于 np.frompyfunc()，根据命名中的“望文生义”原则，这个函数主要是处理来自（from）Python 的函数（func），使之成为能够针对数组进行元素级操作的通用函数。函数的完整结构是 frompyfunc(func, nin, nout)，各参数含义如下。

- func：Python 中定义的函数对象（不是执行函数）。
- nin：一个整数，表示 func 函数所接收的参数个数。
- nout：一个整数，表示 func 函数返回的对象个数。

再看一个示例，以加深理解。

```
In [37]: pow_ufunc = np.frompyfunc(pow, 2, 1)
         a = np.arange(10)
         pow_ufunc(a, a)
Out[37]: array([1, 1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489], dtype=object)

In [38]: a ** a
Out[38]: array([       1,        1,        4,       27,      256,     3125,
                 46656,   823543, 16777216, 387420489])
```

pow()是 Python 中的一个内置函数，In[37]中把它变成 NumPy 的通用函数。当然，事实上在 NumPy 中不用这么麻烦，这里只是借用 pow()来演示如何把一个普通内置函数变成通用函数，并且进一步体会通用函数的特点。

除 np.frompyfunc()外，在 NumPy 中还有一个方法能够将 Python 函数转换为通用函数，如下所示。

```
In [39]: bmi_ufunc2 = np.vectorize(bmi, otypes=[np.float])
         bmi_ufunc2(h, w)
Out[39]: array([ 0.,  1.,  0., -1.,  0., -1.,  0., -1.,  0., -1.])
```

原来的 bmi()函数只能接收“标量”参数，不能接收“矢量”（数组），而 np.vecotrize()的作用就是将 bmi()函数对象矢量化，之后就可以接收数组作为参数了。并且，在矢量化的时候，还能通过 otypes 的值设置返回数组的元素类型。

现在已经能够使用 NumPy 的各种运算符和内置的通用函数，甚至自定义的通用函数进行各种运算了。从理论上讲，这已经足够了，但是 NumPy 并没有仅限于此，为了彰显它给数据处理带来的便利，还内置了常见的统计量的计算方法。

1.5　简单统计应用

NumPy 中提供了很多函数，就好比一个巨大无比的工具箱。当然，本书不可能把所有工具

一一列举，只能重点介绍几个，读者可以通过它们了解工具的基本使用方法。在日后项目实践中，基本的原则还是依靠 Google 进行搜索，然后从诸多备选项中选择合适的。

本节所要介绍的函数是与统计有关的，NumPy 对数据统计的支持很好、很强大，请读者慢慢体会。

1．生成正态分布数据

很多现象都符合正态分布规律——有专门解释其原因的资料，建议读者去查阅。

正态分布（Normal Distribution，也被译作“常态分布”），又叫作高斯分布（高斯，德国数学家、物理学家），是统计学中非常有名的一个函数，表达式为：

$$f(x)=\frac{1}{\sqrt{2\pi}\sigma}\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

在 NumPy 中有一个名为 random 的模块，里面包含若干生成随机数据的函数，其中 normal 就是专门用来生成符合正态分布规律的随机数字，完整的函数表达式是 numpy.random.normal (loc=0.0, scale=1.0, size=None)。

- loc：浮点数，分布的平均值，即公式中的 μ。
- scale：浮点数，分布的标准差，即公式中的 σ（对应于分布的宽度，scale 越大，越矮胖；scale 越小，越瘦高）。
- size：整数或者整数元素的元组，输出的数据个数，默认为 None。

```
In [1]: import numpy as np
        mu, sigma = 0.0, 1.0
        g = np.random.normal(loc=mu, scale=sigma, size=1000)
```

数组 g 的元素是由浮点数构成的，这些浮点数按照正态分布的规律分布。除这种做法外，还有另外一个可以获得随机数的函数 np.random.randn(size)，利用这个函数所得到的随机数是符合 loc=0.0、scale=1.0 条件的正态分布——称之为“标准正态分布”。

```
In [2]: g1 = np.random.randn(1000)
```

g 和 g1 都是符合标准正态分布的，当然其中的元素由于是随机生成的，不都是相等的。

接下来，我们编造一组考试成绩——考试成绩的分布规律应该符合正态分布，否则意味着可能有大量作弊嫌疑，或者考试题目异常。

```
In [3]: g = (g - g.min()) / 10
        phy = np.round(((100.1 - 20) * g + 20), 1)
        g1 = (g1 - g1.min()) / 10
        mat = np.round(((100.1 - 20) * g1 + 20), 1)
```

在 In[3]中，我们使用 g 和 g1 分别构建了两组数据 phy 和 mat，用它们来代表某次考试的物理、数学成绩。接下来就是用 NumPy 中提供的函数处理这些考试成绩。

2．简单的统计函数

表 1-5-1 列出了 NumPy 中常用的并且简单的统计函数，能够得到常用的数据统计量。

表 1-5-1　简单统计函数

函　数	说　明
np.mean，np.average	计算平均值、加权平均值
np.var	计算方差
np.std	计算标准差
np.min，np.max	计算最小值、最大值
np.argmin，np.argmax	返回最小值、最大值的索引
np.ptp	计算全距，即最大值和最小值的差
np.percentile	计算百分位在统计对象中的值
np.median	计算统计对象的中值
np.sum	计算统计对象的和

按照通常的教育统计要求，对前面已经得到的 phy 数据进行如下处理。

```
In [4]: np.mean(phy)
Out[4]: 45.921500000000002

In [5]: np.std(phy)
Out[5]: 7.9199045290962946

In [6]: np.min(phy), np.max(phy)
Out[6]: (20.0, 67.799999999999997)

In [7]: np.percentile(phy, 80)
Out[7]: 52.799999999999997

In [8]: np.median(phy)
Out[8]: 45.849999999999994
```

读者如果也执行上述操作，得到的结果与之不同，这是正常的。如果相同，纯属巧合。

以上对 phy 做了基本统计，从上述操作中不难看出，没有什么复杂的操作。当然，NumPy 不仅能够对一维数组的数据进行统计，还能够对多维数组的数据进行统计。下面就演示二维数组的情况。

```
In [9]: marks = np.vstack((phy, mat))
        marks
Out[9]: array([[ 66.4,  41. ,  55.6, ...,  58.8,  34.5,  40.1],
               [ 36.3,  53.7,  49. , ...,  48.9,  45.4,  59.9]])  #显示方面 Jupyter 自动做了省略
```

假设 phy 和 mat 的成绩都是一一对应的，也就是相同索引的成绩是同一个学生的，那么 In[9] 得到的 marks 就是包含 1000 个学生的物理（phy）和数学（mat）成绩的二维数组，对这个数组进行如下操作，在每行操作后面通过注释说明操作的目的。

```
In [10]: np.mean(marks, axis=1)     #沿 1 轴方向计算平均值，分别得到 phy 和 mat 的平均值
Out[10]: array([ 45.9215,  48.124 ])

In [11]: np.mean(marks, axis=0)    #沿 0 轴方向计算平均值，得到每个人两个学科的平均值
Out[11]: array([ 50.6 ,  46.5 ,  47.7 ,  49.25,  48.7 ,  47.35,  36.5 ,  46.65,
                ……    #因为篇幅所限，省略部分元素
```

```
                47.05,  49.7 ,  49.8 ,  49.25,  48.95,  52.85,  49.9 ,  54.6 ])

In [12]: np.std(marks, axis=1)    #沿 1 轴方向计算标准差
Out[12]: array([ 7.91990453,  7.85784474])
```

对于每个实施统计的函数，除能够对所有数据进行操作外，如果要声明 axis 的值（指明是哪一个轴），则针对该轴方向的数据实施操作。这种效果类似在电子表格中按照行或者列进行计算。

我们已经知道，如果以布尔类型的数组作为下标，则可以根据要求选出相应的元素。比如要找出 phy 中值超过平均值的元素。

```
In [13]: phy[phy > np.mean(phy)]
Out[13]: array([ 55.5,  48.1,  51.9,  59.8,  51.3,  46.9,  62.6,  52.2,  52. ,
                ……    #因为篇幅所限，省略部分元素
                55.2,  48.7,  53.4])
```

这类学生大多前途无量，因为物理成绩好——一个前物理老师说的。

除这种方法外，还有一个非常好用的 np.where()函数，它其实是面向数组的条件语句的三元操作（A if condition_True else B，如果读者对此陌生，请参阅《跟老齐学 Python：轻松入门》有关章节的内容）。例如，对于 phy 和 mat 两个数组，我们要比较相应值的大小，如果 phy 中的元素大于 mat 中的元素，就返回 1，否则返回–1。

```
In [14]: np.where(phy > mat, 1, -1)
Out[14]: array([-1, -1,  1, -1, -1, -1,  1, -1, -1,  1,  1,  1,  1, -1, -1, -1,  1,
               ……    #因为篇幅所限，省略部分元素
```

当然，也可以只写条件，那么返回的就是由符合条件的元素索引组成的数组。

```
In [15]: np.where(phy > phy.mean())
Out[15]: (array([  2,   3,  10,  11,  14,  15,  16,  17,  18,  20,  21,  23,  25,
                ……    #因为篇幅所限，省略部分元素
                994, 996, 997, 999]),)
```

np.where()实现了三元操作，而另外一个函数 np.select()则能够实现多分支判断和选择。继续用 phy 这个数组，下面打算给每个人的分数评一个等级，规则是：

- 高于平均分 20 分的为 A；
- 高于平均分且低于平均分 20 分的为 B；
- 其他为 C。

```
In [16]: grand_a = np.mean(phy) + 20
         grand_b = np.mean(phy)
         np.select([phy >= grand_a, np.logical_and(phy >= grand_b, phy < grand_a)],
                   ['A', 'B'], default='C')
Out[16]: array(['C', 'C', 'B', 'B', 'C', 'C', 'C', 'C', 'C', 'C', 'B', 'B', 'C',
                ……    #因为篇幅所限，省略部分元素
```

还有一个函数 np.piecewise()，也能实现按条件筛选并输出指定结果的操作。建议读者使用“np.piecewise?”方式查看文档说明。

```
In [17]: np.piecewise(phy, [phy > grand_a, np.logical_and(phy >= grand_b, phy < grand_a)],
                      [100, 60])
```

```
Out[17]: array([ 100.,    0.,   60.,   60.,   60.,    0.,  100.,   60.,   60.,
                 ……    #因为篇幅所限，省略部分元素
```

从数学角度看，可以把 np.select()和 np.piecewise()理解为对分段函数的操作，这里展示的是最简单的应用，如果读者在工程实践中需要更复杂的操作，一定要查看相关文档。你我都没有“超强大脑”，也没有必要锻炼过目不忘的记忆力，但是我们需要掌握研究方法，具有快速学习的能力，这才是立于不败之地的根本。

1.6　矩阵

矩阵，是一个数学概念。

二维数组是我们经常用到的，因为二维数组的样式跟矩阵一样，所以有人把二维数组也称为矩阵。在 NumPy 中有专门针对纯粹矩阵的定义和函数，这说明二维数组和矩阵还是有一些区别的。

1．创建矩阵

在 NumPy 中，矩阵是 ndarray 的子类，因此每个矩阵首先继承了数组的属性和方法，然后才有该矩阵自己特有的属性和方法。

```
In [1]: import numpy as np
        np.matrix?
Init signature: np.matrix(data, dtype=None, copy=True)
Docstring:
matrix(data, dtype=None, copy=True)

Returns a matrix from an array-like object, or from a string of data.
A matrix is a specialized 2-D array that retains its 2-D nature
through operations.  It has certain special operators, such as ``*``
(matrix multiplication) and ``**`` (matrix power).

In [2]: np.mat?
Signature: np.mat(data, dtype=None)
Docstring:
Interpret the input as a matrix.

Unlike `matrix`, `asmatrix` does not make a copy if the input is already
a matrix or an ndarray.  Equivalent to ``matrix(data, copy=False)``.
```

创建矩阵的方法有 np.mat()和 np.matrix()，上面截取了两个方法的官方文档部分内容，从中读者能够初步理解两个方法的相同点和不同点，以及分别使用这两个方法创建矩阵的参数要求。

```
In [3]: A = np.mat([1,2,3])
        A
Out[3]: matrix([[1, 2, 3]])

In [4]: A.shape
Out[4]: (1, 3)

In [5]: a = np.array([1,2,3])
```

```
        a
Out[5]: array([1, 2, 3])

In [6]: a.shape
Out[6]: (3,)
```

In[3]所创建的 $\boldsymbol{A}$ 是一个矩阵，In[5]所创建的 a 是我们所熟悉的数组。注意观察两者所传入的参数对象，虽然都是[1, 2, 3]，但 Out[4]得到的矩阵 $\boldsymbol{A}$ 的形状和 Out[6]中数组 a 的形状不同，这就是矩阵和数组的一个区别：矩阵必须是二维的，而数组可以是一、二、三等维度。

除用 In[3]所示方式创建矩阵外，还可以用字符串作为参数创建矩阵。

```
In [9]: B = np.mat('1 2 3; 4 5 6; 7 8 9')
        B
Out[9]: matrix([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

或者以数组为参数创建矩阵。

```
In [10]: C = np.mat(a)
         C
Out[10]: matrix([[1, 2, 3]])
```

还能够以矩阵为参数创建新的矩阵——使用 np.mat()。

```
In [11]: M1 = np.mat(np.eye(2))
         M2 = 2 * M1
         D = np.bmat('M1 M2; M2 M1')
         D
Out[11]: matrix([[ 1.,  0.,  2.,  0.],
                 [ 0.,  1.,  0.,  2.],
                 [ 2.,  0.,  1.,  0.],
                 [ 0.,  2.,  0.,  1.]])
```

矩阵也是对象（以上所创建的矩阵可以看作是类 numpy.matrix 的实例对象），具有一些属性和方法，有的也是继承自数组对象，所以此处从略，读者可以查阅有关资料（比如 https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html）了解详情。

跟数组相比较，矩阵的乘法运算独树一帜，请读者特别小心。

2．乘法

矩阵与标量的乘法和前面已经学习过的数组与标量的乘法一样，不再赘述。下面重点讲解矩阵与矩阵的乘法。

我们已经熟知，两个形状相同（必要时做广播）的数组相乘，其元素对应相乘，从而得到一个相同形状的新数组（例如 In[12]）。而对于矩阵，虽然它是数组的子类，乘法却不同于数组（例如 In[13]）。

```
In [12]: a = np.arange(1, 10).reshape((3,3))
         b = np.array([[1,0,1],[0,1,1],[1,1,0]])
         a * b
Out[12]: array([[1, 0, 3],
                [0, 5, 6],
```

```
                [7, 8, 0]])

In [13]: A = np.mat(a)
         B = np.mat(b)
         A * B
Out[13]: matrix([[ 4,  5,  3],
                 [10, 11,  9],
                 [16, 17, 15]])
```

比较 Out[12]和 Out[13]的结果，同样使用了乘法符号“*”，但二维数组和矩阵是有差别的。对于矩阵乘法的理解，推荐读者阅读“知乎”上的“矩阵乘法的本质是什么”（https://www.zhihu.com/question/21351965#answer-7804963）。

因为二维数组和矩阵雷同，所以我们也希望通过对二维数组的操作，实现类似矩阵乘法的运算。NumPy 也这么“想”了，它提供了一个函数 numpy.dot()。

```
In [14]: np.dot(a, b)
Out[14]: array([[ 4,  5,  3],
                [10, 11,  9],
                [16, 17, 15]])
```

np.dot(a, b)和“A * B”的结果是一样的，矩阵的 In[13]运算与二维数组的 In[14]运算等效，并没有额外地增加运算成本，但是 In[13]在视觉上更直观，更符合数学中的习惯。关于 np.dot()函数，后面会有详细介绍。

3．基本操作

前面所创建的矩阵，当然也是对象，所以就有相应的属性和方法，通过它们实现数学中对矩阵的常用操作。不过，因为矩阵是数组的子类，所以对数组成立的操作对矩阵也成立（再次提醒，二维数组可以看作矩阵来处理）。这里介绍矩阵中常用的几个概念。

```
In [15]: A = np.mat('3 0 1; 0 2 2; 1 2 4')
Out[15]: matrix([[3, 0, 1],
                 [0, 2, 2],
                 [1, 2, 4]])
```

创建一个矩阵 $\boldsymbol{A}$，但这个矩阵不是随随便便创建的，在学习后续内容的过程中，请读者一定要努力回忆（如果忘记了，最好进行复习或者学习）在数学中已经学习过的关于矩阵的知识。

转置是矩阵中常用的操作之一，通常我们使用一种非常简单的方式实现。

```
In [16]: A.T
Out[16]: matrix([[3, 0, 1],
                 [0, 2, 2],
                 [1, 2, 4]])
```

此外，还可以使用 A.getT()实现同样的操作。

建议读者执行 dir(A)，翻阅矩阵对象的属性和方法，然后继续阅读。

另外，还有逆矩阵。注意，不是任何一个矩阵都有逆矩阵的，所以要使用刚刚建立的矩阵 $\boldsymbol{A}$ 来演示，因为它是依据某矩阵可以有逆矩阵的条件创建的，当然，在实践中，读者可以对任何矩阵计算其逆矩阵。如果能够计算出来，就说明有逆矩阵；如果报错，就说明该矩阵无逆矩阵——数学系的同学对此表示不理解，为什么不先证明是否可逆——物理系的同学不这么想。

```
In [17]: A.I
Out[17]: matrix([[ 0.4,  0.2, -0.2],
                 [ 0.2,  1.1, -0.6],
                 [-0.2, -0.6,  0.6]])

In [18]: A * A.I    #矩阵与其逆矩阵的乘积是单位矩阵
Out[18]: matrix([[ 1.,  0.,  0.],
                 [ 0.,  1.,  0.],
                 [ 0.,  0.,  1.]])

In [19]: np.mat(np.eye(4))    #创建单位矩阵的方法
Out[19]: matrix([[ 1.,  0.,  0.,  0.],
                 [ 0.,  1.,  0.,  0.],
                 [ 0.,  0.,  1.,  0.],
                 [ 0.,  0.,  0.,  1.]])
```

除可以通过矩阵属性完成一些操作外，NumPy 中有一个模块是专门针对线性代数而言的，这个模块的名字叫作 linalg（linear algebra，线性代数）。numpy.linalg 可以完成更多关于矩阵的计算，比如求特征值、解线性方程（组）、计算行列式等。其功能强大，内容广博。

4．斐波那契数列

在《跟老齐学 Python：轻松入门》一书中，已经列举了几种计算斐波那契数列的方法，在这里我们依然要计算这个数列，只不过这次要使用矩阵。

```
In [20]: F = np.mat([[1, 1], [1, 0]])
         F
Out[20]: matrix([[1, 1],
                 [1, 0]])

In [21]: f8 = F ** 7
         f8
Out[21]: matrix([[21, 13],
                 [13,  8]])

In [22]: f8[0, 0]
Out[22]: 21
```

In[21]是计算斐波那契数列的关键，要获得数列中第 8 个数，运算结果得到了一个矩阵，然后取矩阵第一个元素（In[22]），即为所需的数。这样做的理论根据是什么？请阅读 http://www.maths.dur.ac.uk/~dma0rcj/PED/fib.pdf 中的有关证明。

本节简要地讲解了矩阵及其与乘法相关的事项，这些只是矩阵的一小部分。矩阵是线性代数中的重要内容，也是数据分析中经常用到的。所以，读者完成本节学习之后，不要局限于这些内容，而要尽可能依据官方文档和有关网络资料拓展自己的知识范围。本节对矩阵的讲述，只能是“领进门”，并期望能“授之以渔”，以后还要靠读者自己不断“修行”。

1.7 矢量运算

本书将矢量作为单独一节阐述，这在众多关于 NumPy 或数据分析的书中是少见的，之所以

如此，是因为笔者在曾经的一段学习和工作中深切感受到矢量的重要性。至于读者，是否有如笔者那样的感受，要看出身了——大物理系的、包括向往大物理系的同学，都要重视本节，这是专为你而写的。

1．矢量和标量

所谓标量，即只有大小没有方向的量，在 NumPy 中通常用单个数值表示；所谓矢量，即有大小和方向的量，在 NumPy 中可以用数组表示。比如图 1-7-1 所表示的是三维坐标系中的一个矢量 $\boldsymbol{a}$，在这个坐标系中，它可以表示为 $\boldsymbol{a}=a_x\vec{i}+a_y\vec{j}+a_z\vec{k}$，其中 a_x、a_y、a_z 分别为矢量 $\boldsymbol{a}$ 的长度（大小）在各相应坐标轴上的投影长度（大小）。

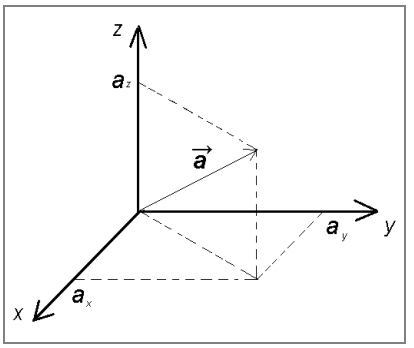


图 1-7-1　矢量

在 NumPy 中，如果用数组来表示矢量 $\boldsymbol{a}$，则通常使用一维数组，即 numpy.array([a_x, a_y, a_z])，特别注意各个元素的顺序。拓展一下，如果某个矢量是在 n 维坐标系中，则也可以用类似的一维数组表示。

理解了如何表示矢量之后，就可以研究矢量与矢量、矢量与标量之间的运算。以下运算，有一部分就是前文所学知识的应用。

- 矢量与标量的运算，就是标量与数组中的各个元素之间的运算，读者应该很容易理解，就不做过多解释了。
- 矢量与矢量的加法（减法）运算，就是两个数组相应元素的运算。为了跟实际的情形结合，举一个力的合成的例子。比如平面中有两个力，一个力的大小是 10N、与水平方向夹角是 30°，另一个力的大小是 12N、与水平方向夹角是 90°，计算两个力的合力。首先用矢量的方式把两个力分别表示出来，即 $\boldsymbol{F_1}=10\cos30^\circ\vec{i}+10\sin30^\circ\vec{j}$、$\boldsymbol{F_2}=0\vec{i}+12\vec{j}$，具体计算过程如下。

```
In [1]: import numpy as np
        import math
        f1 = np.array([10*math.cos(math.pi/6), 10*math.sin(math.pi/6)])    #①
        f2 = np.array([0, 12])
        f = f1 + f2    #②
        f
Out[1]: array([  8.66025404,  17.        ])

In [2]: np.sqrt(np.sum(f**2))
Out[2]: 19.078784028338912
```

```
In [3]: np.linalg.norm(f)
Out[3]: 19.078784028338912
```

In[1]的①中使用 math.cos()计算余弦，而没有使用 np.cos()，这是因为对于单个数值的运算，NumPy 并没有优势，还是使用 Python 中的函数好。

用 f1 和 f2 两个数组表示两个力，计算合力即两个矢量相加，用 In[1]的②完成矢量加法（如果计算矢量减法，也同此），最终得到的 f 即为计算结果。如果用矢量表示这个合力，则为 $\boldsymbol{F}=8.7\vec{i}+17\vec{j}$（对小数位数进行了取舍）。要计算合力大小，可以使用 In[2]或者 In[3]。

对于矢量之间的运算，除加法（减法）外，还有乘法（除法），但后者较为复杂，请耐心、细心阅读下文。

2．标量积

标量积（Scalar Product）是两个矢量相乘的一种方式和结果（此处关于标量积的数学内容，参考高等教育出版社《数学手册》1979 年第一版“第八章矢量算法与场论初步 • 张量算法与黎曼几何初步”中的相关讲述，后续矢量积和张量积亦同此），也被称为数量积、点积（Dot Product）、内积（Inner Product）。顾名思义，按照这种方式两个矢量相乘的结果是一个标量。

（1）代数理解

假设有两个矢量 $\boldsymbol{a}=[a_1,a_2,a_3,\cdots a_n]$ 和 $\boldsymbol{b}=[b_1,b_2,b_3,\cdots b_n]$，它们的点积定义为：

$$\boldsymbol{a}\cdot\boldsymbol{b}=\sum_{i=1}^{n}a_ib_i=a_1b_1+a_2b_2+\cdots+a_nb_n$$

或者

$$\boldsymbol{a}\cdot\boldsymbol{b}=\left|\boldsymbol{a}\boldsymbol{b}^{\mathrm{T}}\right|$$

其中，$\boldsymbol{b}^{\mathrm{T}}$ 是矢量 $\vec{b}$ 的转置，而 $\left|\boldsymbol{a}\boldsymbol{b}^{\mathrm{T}}\right|$ 是 $\boldsymbol{a}\boldsymbol{b}^{\mathrm{T}}$ 的行列式。

（2）几何理解

以平面坐标系中的两个矢量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 为例，它们的夹角是 θ，则

$$\boldsymbol{a}\cdot\boldsymbol{b}=|\boldsymbol{a}||\boldsymbol{b}|\cos\theta$$

标量积的几何理解如图 1-7-2 所示。

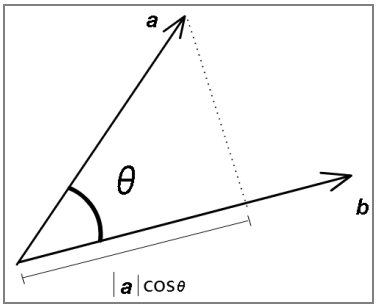


图 1-7-2　标量积的几何理解

为了理解标量积的运算，还是从一个似曾相识的实例开始。

力 F 作用在水平面上的一个物体上，大小为 6N，与水平方向的夹角为 30°（斜向上），物体向前移动了 10m（如图 1-7-3 所示），求这个力所做的功是多少。

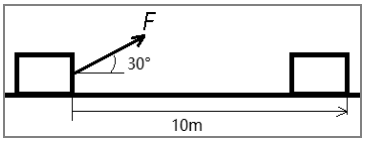


图 1-7-3　标量积运算实例

这是一道很简单的高中物理题目。

这里，我们要使用 NumPy 提供的标量积方法进行计算。

在物理学中，对力做的功的定义为 $W = \boldsymbol{F} \cdot \boldsymbol{S}$，这个定义说明功是力与位移的标量积，力和位移毫无疑问都是矢量，根据题目已知条件得：$\boldsymbol{F} = 6\cos 30^\circ \vec{i} + 6\sin 30^\circ \vec{j}$，$\boldsymbol{S} = 10\vec{i} + 0\vec{j}$。由此可以创建如下矢量，即一维数组。

```
In [4]: f = np.array([6*math.cos(math.pi/6), 6*math.sin(math.pi/6)])
        s = np.array([10, 0])
        np.dot(f, s)
Out[4]: 51.96152422706632
```

最终得到力做的功是 51.96J。

np.dot()是 NumPy 中对数组进行“标量积”运算的函数，In[4]所创建的两个表示矢量的 f 和 s 的数组都是一维数组。当参数是一维数组的时候，np.dot()函数所得结果即两个矢量的标量积。

读者可以在结合对标量积的“代数理解”的计算过程中理解 np.dot()的计算及其结果。

np.dot()函数对于一维数组（矢量）而言进行的是标量积运算，但是当参数可以被看作是矩阵的二维数组时，则按照矩阵的计算方法完成运算。对此官方文档有详细描述，请阅读下面从文档中截取的部分内容。

```
dot(a, b, out=None)

Dot product of two arrays.

For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner
product of vectors (without complex conjugation). For N dimensions it is a sum product
over the last axis of `a` and
the second-to-last of `b`::
    dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

虽然在上一节中已经出现过使用 np.dot()对两个二维数组进行运算的示例，但这里还要重复展示，一来温故知新，二来引导读者将其与矩阵乘法对照，认识到两者是一样的，并进一步理解二维数组可以被当作矩阵的含义。

```
In [5]: c = np.array([[1, 2], [3, 4]])
        d = np.array([[5, 6], [7, 8]])
        np.dot(c, d)
Out[5]: array([[19, 22],
               [43, 50]])
```

对于两个矩阵的标量积，其计算过程如下。

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\cdot\begin{bmatrix}5 & 6\\ 7 & 8\end{bmatrix}=\begin{bmatrix}1\times5+2\times7=19 & 1\times6+2\times8=22\\ 3\times5+4\times7=43 & 3\times6+4\times8=50\end{bmatrix}=\begin{bmatrix}19 & 22\\ 43 & 50\end{bmatrix}$$

在 NumPy 中还有一个名为 inner 的函数 numpy.inner()，貌似是专门要对应“内积”这个名称似的，也能够完成标量积运算。比如用于计算“力做功”的问题：

```
In [6]: np.inner(f, s)
Out[6]: 51.96152422706632
```

那么，这个 np.inner()和 np.dot()有区别吗？从刚才的计算结果推测，至少一维数组为参数的时候没有区别。其他维度的数组作为参数会怎么样呢？还是先看看文档再说。

```
inner(a, b)

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher
dimensions a sum product over the last axes.
```

比较 np.dot()和 np.inner()文档说明及刚才的实际操作可知，对于一维数组（矢量），两个函数都是计算标量积。而对于更高维数的数组，比如二维数组，两个函数就开始有区别了。其中，np.dot()在前面已经说明，它执行的计算过程就是矩阵的乘法；np.inner()按照文档中说的是“a sum product over the last axes”，下面通过一个示例来理解这句话。

```
In [7]: np.inner(c, d)
Out[7]: array([[17, 23],
               [39, 53]])
```

还是用下面的方式展现 np.inner(c, d)的运算流程：

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\cdot\begin{bmatrix}5 & 6\\ 7 & 8\end{bmatrix}=\begin{bmatrix}1\times5+2\times6=17 & 1\times7+2\times8=23\\ 3\times5+4\times6=39 & 3\times7+4\times8=53\end{bmatrix}=\begin{bmatrix}17 & 23\\ 39 & 53\end{bmatrix}$$

对于二维数组，1 轴是最后轴（last axes），np.inner()将两个数组 1 轴上的元素分别对应相乘并求和，得到最终的数组。

请读者对比 np.dot()的计算流程，其差别显而易见。

NumPy 提供了两个针对矢量计算标量积的函数，虽然对于一维数组而言两者无差别，但对于多维数组则各有各的用途。

矢量相乘除标量积外，还有矢量积。

3．矢量积

矢量积（Vector Product），又可以称为叉积（Cross Product）或外积（Outer Product），其运算结果是一个矢量，并且矢量方向与原来两个矢量所构成的平面垂直。

对于矢量积，依然可以从代数、几何两个角度理解。

- 代数理解

我们以三维坐标系中的两个矢量为例，即 $\boldsymbol{a}=a_x\vec{i}+a_y\vec{j}+a_z\vec{k}$ 和 $\boldsymbol{b}=b_x\vec{i}+b_y\vec{j}+b_z\vec{k}$，则：

$$\boldsymbol{a}\times\boldsymbol{b}=\begin{bmatrix}\vec{i} & \vec{j} & \vec{k}\\ a_x & a_y & a_z\\ b_x & b_y & b_z\end{bmatrix}=(a_yb_z-a_zb_y)\vec{i}+(a_zb_x-a_xb_z)\vec{j}+(a_xb_y-a_yb_x)\vec{k}$$

- 几何理解

还是以平面坐标系中的两个矢量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 为例，它们的夹角是 θ，如果对这两个矢量计算矢量积，则结果的大小等于这两个矢量为相邻两边的平行四边形的面积，即 $\boldsymbol{a}\times\boldsymbol{b}=|\boldsymbol{a}||\boldsymbol{b}|\sin\theta$。

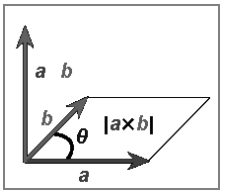


图 1-7-4　矢量积的几何理解

NumPy 中提供了 numpy.cross()函数，是专门用来计算矢量积的。为了讲解这个函数的应用，还是看一个曾经很熟悉的题目。

已知平面直角坐标系中有三个点 $A(1, 1)$、$B(5, 2)$、$C(2, 3)$，求以这三个点为顶点的三角形的面积。

应用中学数学的知识能够解决这个问题，但这里我们使用矢量积来计算这个三角形的面积。

若以 A 点为起始点，连接 AB 和 AC，则分别构成了两个向量，即 $\boldsymbol{AB}=4\vec{i}+1\vec{j}$，$\boldsymbol{AC}=1\vec{i}+2\vec{j}$，转换为一维数组表示该向量，并计算叉积。

```
In [8]: ab = np.array([4, 1])
        ac = np.array([1, 2])
        s = np.cross(ab, ac) / 2
        s
Out[8]: 3.5
```

np.cross(ab, ac)计算得到的结果是 AB、AC 为相邻边的平行四边形的面积（见前面对矢量积的几何理解），三角形面积为其一半。

除这种应用外，在物理中矢量积用途广泛，比如带电体在磁场中运动时所受到的洛伦兹力 $\boldsymbol{F}=q\boldsymbol{v}\times\boldsymbol{B}$，显然这里的速度和磁感应强度都是矢量，它们之间所做的就是矢量积运算。

```
In [9]: v = np.array([1,2,3])
        b = np.array([9, 8, 7])
        np.cross(v, b)
Out[9]: array([-10,  20, -10])
```

Out[9]的结果就显示了 $\boldsymbol{v}$ 和 $\boldsymbol{b}$ 两个矢量的矢量积的大小和方向，它也是一个矢量（在三维坐标系内，其结果表示为 $-10\vec{i}+20\vec{j}-10\vec{k}$）。按照对矢量积的代数理解，将其计算过程进行剖析：

$$\begin{vmatrix}i & j & k\\ 1 & 2 & 3\\ 9 & 8 & 7\end{vmatrix}=(2\times7-3\times8)\vec{i}+(3\times9-1\times7)\vec{j}+(1\times8-2\times9)\vec{k}=-10\vec{i}+20\vec{j}-10\vec{k}$$

np.cross()的参数不仅可以是一维数组（矢量），还可以是多维数组，比如：

```
In [10]: np.cross(c, d)    #c 和 d 是 In[5]中创建的数组
Out[10]: array([-4, -4])
```

除 np.cross()外，还有一个 np.outer()，这个函数在 NumPy 中常被说成是计算“外积”，显然会让人联想到“矢量积”的别称“外积”，这两个概念着实有点让人感觉迷茫。

请认真看文档！下面是两个函数的部分文档内容，请读者静下心来，慢慢读一读。

```
Signature: np.cross(a, b, axisa=-1, axisb=-1, axisc=-1, axis=None)
Docstring:
Return the cross product of two (arrays of) vectors.

The cross product of `a` and `b` in :math:`R^3` is a vector perpendicular to both `a`
and `b`.  If `a` and `b` are arrays of vectors, the vectors are defined by the last axis
of `a` and `b` by default, and these axes can have dimensions 2 or 3.  Where the dimension
of either `a` or `b` is 2, the third component of the input vector is assumed to be zero
and the cross product calculated accordingly.  In cases where both input vectors have
dimension 2, the z-component of the cross product is returned.
```

本书继承了笔者编著的其他两本书的风格，坚决不全文翻译文档。当下，随着机器翻译技术的进步，有人宣称学英文无用，如果这样，恭请体验。再看一下 np.outer()的部分文档说明。

```
Signature: np.outer(a, b, out=None)
Docstring:
Compute the outer product of two vectors.

Given two vectors, ``a = [a0, a1, ..., aM]`` and ``b = [b0, b1, ..., bN]``, the outer
product [1]_ is::

  [[a0*b0  a0*b1 ... a0*bN ]
   [a1*b0    .
   [ ...          .
   [aM*b0            aM*bN ]]
```

特别建议读者查看两个函数的完整文档内容。

从上述文档信息中不难得知，np.outer()的参数对象是一维数组表示的矢量，如果不是一维数组，它会自动将其展平为一维数组。

```
In [11]: v
Out[11]: array([1, 2, 3])

In [12]: b
Out[12]: array([9, 8, 7])

In [13]: np.outer(v, b)
Out[13]: array([[ 9,  8,  7],
                [18, 16, 14],
                [27, 24, 21]])
```

v 和 b 是 In[9]中创建的两个一维数组，按照文档描述，np.outer(v, b)的计算过程如下：

$$[[1\times 9=9 \quad 1\times 8=8 \quad 1\times 7=7],$$

$$[2\times9=18 \quad 2\times8=16 \quad 2\times7=14],$$
$$[3\times9=27 \quad 3\times8=24 \quad 3\times7=21]]$$

通过上面的展示，读者理解 np.outer()的计算方法即可。

显然，np.cross()和 np.outer()的差别还是不小的。

4．张量积

函数 np.tensordot()通常被翻译为“张量积”，在学习这个函数之前，请读者回忆数学中对张量的说明（当然，如果忘记了，应该找资料复习一番），这里引用笔者那本古老的《数学手册》中的说明：张量概念是矢量和矩阵概念的推广。标量是零阶张量，矢量是一阶张量，矩阵是二阶张量，而三阶张量好比“立体矩阵”。

这里的重点还是学习 np.tensordot()函数，从最权威的官方文档开始。

```
Signature: np.tensordot(a, b, axes=2)
Docstring:
Compute tensor dot product along specified axes for arrays >= 1-D.

Given two tensors (arrays of dimension greater than or equal to one), `a` and `b`, and
an array_like object containing two array_like objects, ``(a_axes, b_axes)``, sum the
products of `a`'s and `b`'s elements (components) over the axes specified by ``a_axes``
and ``b_axes``. The third argument can be a single non-negative integer_like scalar, ``N``;
if it is such, then the last ``N`` dimensions of `a` and the first ``N`` dimensions of
`b` are summed over.
```

建议仔细地阅读上面的文档。

阅读文档之后，或许还是似懂非懂，那就请继续看下面的示例，相信读者会有所悟。

```
In [14]: a = np.arange(6).reshape(2, 3)
         b = np.arange(9).reshape(3, 3)
         np.dot(a, b)
Out[14]: array([[15, 18, 21],
                [42, 54, 66]])

In [15]: np.tensordot(a, b, axes=([1], [0]))
Out[15]: array([[15, 18, 21],
                [42, 54, 66]])
```

a 和 b 分别是二维数组，对它们进行 In[14]的操作，就是按照标量积（点积）进行运算，即前面已经介绍的矩阵乘法，Out[14]为标量积（点积）的结果。正如我们已知的标量积（点积）规则，对于 a 和 b 而言，就是 a 的 1 轴方向上的各个元素依次与 b 的 0 轴方向上的各个元素相乘并求和，最终得到 Out[14]的数组。

从前面的文档可知，np.tensordot()能够规定哪些轴参与标量积（点积）计算，那么就在 In[15]中的语句中声明 axes=([1], [0])，意思是 a 的 1 轴、b 的 0 轴参与标量积计算。最终得到的结果和 In[14]的结果是一样的。

通过上面的示例，读者或许理解了 np.tensordot()大概的使用方法。但是，在什么情况下需要规定参与运算的轴呢？我们知道，并不是任意两个矩阵（数组）都可以进行标量积运算的，或者实际需求中不需要所有的数据都参与计算，比如下面的示例。

```
In [16]: a = np.random.randint(2, size=(2, 6, 5))
         b = np.random.randint(2, size=(3, 2, 4))
```

对于这两个数组，不能直接使用 np.dot()（读者自行测试检验），那么就要选择特定轴上的元素参与标量积运算。通过对形状的观察，我们选择 a 的 0 轴元素和 b 的 1 轴元素，从而可以推断出标量积的结果数组的维度，推算流程如下：

$$\begin{bmatrix} a.shape \\ (2,6,5) \end{bmatrix}, \begin{bmatrix} b.shape \\ (3,2,4) \end{bmatrix} \rightarrow \begin{matrix} a:[0], b:[1] \\ 2\ cut\ down \end{matrix} \rightarrow \begin{matrix} np.tensordot().shape \\ (6,5,3,4) \end{matrix}$$

下面就按照刚才对轴的选择使用 np.tensordot()函数进行计算，并验证结论是否符合上述推算。

```
In [17]: np.tensordot(a, b, axes=([0], [1])).shape
Out[17]: (6, 5, 3, 4)
```

完美！至于 np.tensordot(a, b, axes=([0], [1]))的结果，请读者自己计算，这里就不占用篇幅了。

至此，对文档中的“Compute tensor dot product along specified axes for arrays >= 1-D”这句话是不是已经有所理解了呢？还不尽兴的话，可以再看看下面的示例，这里笔者就不多说了，相信读者能够理解。

```
In [18]: c = np.random.randint(2, size=(2,3,5))
         d = np.random.randint(2, size=(3,2,4))
         np.tensordot(c, d, axes=((0, 1), (1, 0)))
Out[18]: array([[1, 0, 0, 0],
                [2, 0, 0, 1],
                [3, 2, 1, 2],
                [2, 1, 0, 0],
                [2, 2, 1, 2]])

In [19]: np.tensordot(c, d, axes=((0, 1), (1, 0))).shape
Out[19]: (5, 4)
```

看来只要耐心地研究，是能够有所收获的。学习过程就是不断挑战自己的过程。

1.8 综合应用示例

在本节中，我们将借助 NumPy 的函数，对基础知识进行综合应用并加深理解。虽然前面借学习数组和矩阵之便，了解了一些基本的或者说常用的函数，但那些仅仅是沧海一粟，本书也不能将所有函数穷尽，只能根据需要选择部分进行介绍，希望读者能够掌握学习新知识的方法。

1．多项式

在数学中，通常用如下形式表示多项式（Polynomial）。

$$f(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n$$

（1）创建多项式

在 NumPy 中如何表示多项式呢？假设有一个我们很熟悉的一元二次多项式 $f(x) = x^2\text{-}2x\text{+}1$，可以这样表示：

```
In [1]: a = np.array([1, -2, 1])
        p = np.poly1d(a)
```

```
        p
Out[1]: poly1d([ 1, -2,  1])

In [2]: type(p)
Out[2]: numpy.lib.polynomial.poly1d
```

NumPy 中提供了 numpy.poly1d()函数，它以一维数组为参数，创建一个多项式。

将多项式按照项的次数从高到低排列，各项的系数组成一个序列，例如 In[1]中按照 x 的次数从高到低排列后，系数组成的序列就是[1, -2, 1]，然后用这个序列作为参数创建一维数组。假如遇到了如同 $f(x)=x^3-2x+1$ 的形式，则系数组成的序列为[1, 0, -2, 1]，二次项缺失，其系数为 0。代表系数的一维数组创建好之后，就可以使用 np.poly1d()函数创建多项式对象了（如 In[1]所示）。

创建好多项式对象之后，可以给这个多项式的变量传入一个数值，例如计算 $f(1)$，一看就知道应该等于 0。

```
In [3]: p(1)
Out[3]: 0
```

以上是创建一个多项式的一般方法，有时我们需要在上述基础上综合别的技能，创建稍微复杂一些的多项式，比如创建 $f(x)=(x-1)^2+3(x-1)+1$，当然，一种最简单的方式是根据多项式展开的知识，将此多项式展开并合并同类项之后，依据前述方法创建。除此之外，还可以像下面这么做。

```
In [4]: g = np.poly1d([1, 3, 1])
        f = lambda x: g(x-1)    #①
        f(0.0)
Out[4]: -1.0

In [5]: f(1)
Out[5]: 1
```

在 In[4]中创建了一个多项式，注意这里并没有使用数组作为参数，使用的是列表。或者说，创建多项式的参数只要是一个类数组对象即可。然后在 In[4]的①中巧妙地使用了我们熟知的 lambda 函数，实现了变量从 x 到 x-1 的转换。

至此，读者已经初步知道在 NumPy 中如何创建一个多项式了，若要更深入地理解 np.poly1d()函数的用法，当然是在已经入门的基础上阅读其官方文档了——这是反复提及的学习方法。如果没有入门，直接看文档可能糊涂，入门之后看文档则是提升。

```
Init signature: np.poly1d(c_or_r, r=False, variable=None)
Docstring:
A one-dimensional polynomial class.

A convenience class, used to encapsulate "natural" operations on polynomials so that said
operations may take on their customary form in code (see Examples).
#（省略部分后续内容）
```

在 np.poly1d()函数文档的完整形式中，参数还有 r=False 和 variable=None，这两个参数是什么意思？请看如下示例。

```
In [6]: p2 = np.poly1d(a, r=True)
        p2
```

```
Out[6]: poly1d([ 1.,  0., -3.,  2.])

In [7]: print(p2)    #在显示效果上，这里相对 Jupyter 中的显示做了适当美化
        1 x^3 - 3 x + 2

In [8]: p2(1)
Out[8]: 0.0

In [9]: p2(-2)
Out[9]: 0.0
```

使用数组 a 创建一个多项式 p2，不过这次声明了 r=True（如 In[6]所示），r 参数指明 a 中的元素是多项式的系数，还是多项式的根。默认（r=False）是多项式的系数。如果 r=True，则 a 中的元素为多项式 p2 的根。也就是说，利用数组 a 所创建的多项式其实是$(x-1)(x+2)(x-1)=(x-1)^2(x+2)=x^3-3x+2$，所以在 Out[6]中看到的这个多项式系数是符合展开式所显示的。在 In[8]和 In[9]中分别用两个根来检验，结果都是 0。

另外一个参数 variable 用来声明多项式中变量的符号，它接收的是一个字符串。

```
In [10]: p3 = np.poly1d(a, variable="m")
         print(p3)
         1 m^2 - 2 m + 1
```

根据数学知识，多项式还有很多运算。

（2）多项式运算

不同的多项式之间，可以进行加、减、乘、除等运算，以多项式 p 和 p2 为例，演示这些运算。

```
In [11]: print(p)
         print(p2)
         1 x^2 - 2 x + 1
         1 x^3 - 3 x + 2

In [12]: print(p + p2)
         1 x^3 + 1 x^2 - 5 x + 3

In [13]: print(p - p2)
         -1 x^3 + 1 x^2 + 1 x - 1

In [14]: print(p * p2)
         1 x^5 - 2 x^4 - 2 x^3 + 8 x^2 - 7 x + 2
```

读者运用中学时期学过的多项式运算知识，即可理解上述过程。不过，里面没有除法，因为多项式除法有需要特别关注之处——除法总是有点特殊，在 Python 基础知识中也是如此，下面单独演示。

```
In [15]: p2 / p
Out[15]: (poly1d([ 1.,  2.]), poly1d([ 0.]))
```

多项式 $1\,x^3 - 3\,x + 2$ 除以 $1\,x^2 - 2\,x + 1$ 的结果应该是 $x+2$，Out[15]结果中的第一项 poly1d([1, 2])即为商，而 poly1d([0.])是相除的余数，In[15]中两个多项式相除的余数是 0。

此外，对于多项式的运算，还可以使用下面的形式。

```
In [16]: p4 = p2 + [-2, 1]
         p5 = p2 * np.array([3,2,1])
         print(p4)
         print(p5)
         1 x^3 - 5 x + 3    #本行和下一行都是 print()的结果
         3 x^5 + 2 x^4 - 8 x^3 + 1 x + 2

In [17]: p5 / [2, 3, 4]
Out[17]: (poly1d([  1.5   ,  -1.25  ,  -5.125 ,  10.1875]),
         poly1d([ -9.0625, -38.75  ]))
```

多项式的运算，关键是相同次数项的系数之间的运算。既然通过类数组对象（比如[-2, 1]或 np.array([3,2,1])）能够创建多项式，且类数组对象的元素就是多项式系数（r=False），那么也可以把类数组对象和多项式直接运算。NumPy 会将类数组对象看作多项式参与到运算中，比如 In[17]，在得到的结果中，除商外，还有不是 0 的余数。

多项式对象除参与加、减、乘、除运算外，还有微分和积分的运算。

```
In [18]: p.deriv()    #微分
Out[18]: poly1d([ 2, -2])

In [19]: p.integ()    #积分
Out[19]: poly1d([ 0.33333333, -1.        ,  1.        ,  0.        ])

In [20]: p.integ().deriv()    #积分之后再求微分，返回原多项式
Out[20]: poly1d([ 1., -2.,  1.])
```

在数学中，计算多项式根和由根计算多项式系数也是常用到的，于是 NumPy 提供了相应的方法完成这些任务。例如：

```
In [21]: np.roots(p4)   #计算 f(x)= 1 x^3 - 5 x + 3 的根
Out[21]: array([-2.49086362,  1.83424318,  0.65662043])
In [21]: r = np.array([1, -1])
         fr = np.poly(r)
         fr
Out[21]: array([ 1.,  0., -1.])

In [22]: print(np.poly1d(fr))
         1 x^2 - 1
```

In[21]所创建的数组作为多项式的根，通过 np.poly()函数能得到该多项式的系数，在 In[22]中打印出的这个多项式，就是在中学数学中学过的二次函数。

请读者在 Jupyter 交互模式中输入 np.poly，然后按 Tab 键，可以得到如图 1-8-1 所示的结果。我们可以看到所有以 poly 开头的函数，所以不用担心记不住这些方法名称。另外，在这里也可以看到更多跟多项式有关的函数，如果读者有兴趣，可以逐一查阅，了解其概况，以便日后应用。

（3）拟合多项式

物理学为我们提供了发现科学规律的一种有效方法，其流程如下：

① 通过实验或者观测获取数据；

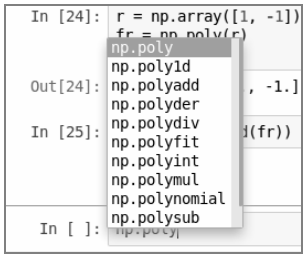


图 1-8-1　查看所有以 poly 开头的函数

② 将所有数据在坐标系（比如二维坐标系）中描点，去除明显无效的数据点；

③ 画一条曲线，使得有效的数据点尽可能均匀分布在曲线两侧或者被曲线连接；

④ 根据经验和已经掌握的数学知识，确定曲线所对应的函数式；

⑤ 对上述结果在新的数据中进行检测和修正，直到满意为止。

这套方法在众多科学、工程中，已经被认为是标准流程了。

我们将上述“画曲线”的过程称之为“拟合”。NumPy 提供了一个对某些数据使用多项式函数进行拟合的方法：np.polyfit()。

为了演示拟合的方法，需要先创建一些数据，我们用下面的方法创建用于拟合的数据。

```
In [23]: x = np.linspace(0, 2*np.pi, 10000)
         y = np.sin(x)
```

x 是由 10000 个数据组成的数组，对应的 y 是用 sin()函数计算出来的，在这里读者假装不知道 x 和 y 的关系。

我们假装已经通过实验获得了两组一一对应的数据 x 和 y，接下来就继续装模作样地寻找这两组数据之间的关系。

```
In [24]: f3 = np.polyfit(x, y, 3)
         f5 = np.polyfit(x, y, 5)
         f7 = np.polyfit(x, y, 7)
```

面对 x 和 y 两组数据，作为训练有素的数据工程师，一看就知道它们之间肯定不符合 1 次和 2 次多项式关系（这建立在对这两类多项式熟悉的基础上），然后进行猜测，或许是更高次的多项式，再用 NumPy 计算一下。如果是 3 次多项式，利用上述数据，得到了多项式 f3（见 In[24]，f3 其实是多项式的系数），依此类推，暂时猜测到 7 次多项式。为什么只猜测奇数次？偶数次不可以吗？当然可以！这里仅仅是一种演示罢了。

上面得到了三个多项式，到底哪一个更准确呢？物理学中的检验方法是将此规律运用到新的实验数据中，新数据更符合哪个函数，则那个函数就更接近真理。但是，在这里我们知道它们真实的函数关系（真理已知），所以可以找一种方法检验一下 f3、f5、f7 中哪一个多项式更接近 sin(x)，那么它就是我们目前找到的最优多项式，或者说是最优拟合函数。

```
In [25]: space3 = np.abs(np.polyval(f3, x) - y)
```

np.polyval()函数相当于合并了多项式运算中的建立多项式和赋值计算两个过程，即：

```
In [26]: pf3 = np.poly1d(f3)
         pf3(x)
```

```
Out[26]: array([-0.203109  , -0.2019105 , -0.20071271, ...,  0.20071271,
                0.2019105 ,  0.203109  ])

In [27]: np.polyval(f3, x)
Out[27]: array([-0.203109  , -0.2019105 , -0.20071271, ...,  0.20071271,
                0.2019105 ,  0.203109  ])
```

In[26]是已经学习过的，In[27]中使用 np.polyval()函数一次性完成前面两个步骤。继续使用 In[25]的方法，计算其余两个多项式与 y 之间的差的绝对值。

```
In [28]: space5 = np.abs(np.polyval(f5, x) - y)
         space7 = np.abs(np.polyval(f7, x) - y)
```

为了查看到底哪个多项式更接近“真理”，一种比较粗糙的方式是比较 space3、space5、space7 三组数据的最大值，距离“真理”的差越小，则越接近 sin(x)。

```
In [29]: np.max(space3), np.max(space5), np.max(space7)
Out[29]: (0.20310899877773878, 0.015943766615314416, 0.00066258287715054642)
```

显然 f7 更接近“真理”。

只是，以上判断方法太粗糙了，而且也不是标准的统计学判断方法。如果能够画图，就更一目了然了。的确如此，只是笔者打算把画图的内容放在后面讲解，请读者继续坚持阅读本书，会看到更精彩的拟合。

在 NumPy 中，除上述对多项式的操作外，还有一个专门的模块 numpy.polynomial，提供了非常丰富的关于多项式的操作方法。建议读者抽出宝贵时间独立学习——运用本书所倡导的方法。

2．解线性方程组

本书 1.6 节曾经提到了 numpy.linalg 模块，这是专门针对线性代数的模块，它提供了解线性方程组的方法。

通常，用下面的方式表示线性方程组。

$$
\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1\\
&a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2\\
&\cdots\cdots\\
&a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m
\end{aligned}
$$

为了求解上面的线性方程组，可以转换为矩阵方式，如下所示：

$$
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1n}\\
a_{21} & a_{22} & \cdots & a_{2n}\\
\vdots & \vdots & \ddots & \vdots\\
a_{m1} & a_{m2} & \cdots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
x_1\\ x_2\\ \vdots\\ x_n
\end{bmatrix}
=
\begin{bmatrix}
b_1\\ b_2\\ \vdots\\ b_n
\end{bmatrix}
$$

通常线性方程组可以用简化方式 $\boldsymbol{Ax} = b$ 表示，其中 $\boldsymbol{A}$ 是 $m \times n$ 的矩阵。对于这种类型的线性方程组，有一个特殊状态，即 $m=n$，这时方程组有唯一确定的解。对于这种状态时的求解，可以使用 numpy.linalg.solve()完成。为了更直观地了解这个函数的使用，假设一个简单的方程组：

$$\begin{cases} 3x_0 + x_1 = 9 \\ x_0 + 2x_1 = 8 \end{cases}$$

按照前面的求解方法，分别创建二维数组 a 和 b，然后使用 numpy.linalg.solve()求解。

```
In [30]: a = np.array([[3, 1], [1, 2]])
         b = np.array([9, 8])、
         x = np.linalg.solve(a, b)
         x
Out[30]: array([ 2.,  3.])
```

最终结果 x 数组的两个元素分别是 x_1 和 x_2。

这是一种比较简单的应用示例，其实线性方程组在实际生产和生活中有很多用途，读者如果有兴趣，可以通过网络搜索，阅读相关资料。

而求解方程组的方法，numpy.linalg.solve()也仅仅是针对 $m=n$ 这种情况的简单方法。除这个函数外，在科学计算专用的库 SciPy 中，还有专门的方法用来解决各种类型的线性方程组问题，有需要的读者可以查阅。

NumPy 是从事数据分析最基本的一个库，博大精深，本章仅就最基本的内容进行介绍。如果本章内容能够让读者对 NumPy 有了基本认识，那么目的已经达到，在真实的项目中，读者完全有能力在此基础上深入研究某个具体的应用。而在具体实践中，除 NumPy 外，还会用到别的知识，比如 Pandas，请看下一章。

第2章

Pandas基础和应用

学完第 1 章，属于“降龙十八掌只练会第一招”的水平，要在数据分析的江湖中行走，还欠火候呢，所以要继续学习后续招式。Pandas 堪称必杀技，值得拥有。

这里所说的 Pandas 不是那个国宝，而是另外一个用于数据分析的 Python 第三方库。Pandas 在 NumPy 的基础上，优化了数据结构，在数据的存储、读取、分割、转换等方面进行了改进，使得操作更简单。因为 Pandas 在数据结构上具有优势，所以有不少关于数据分析的工具在应用中都依赖它。因此，Pandas 也是数据分析的标配。

2.1 常用数据对象

Pandas 提供了三种数据对象，分别是 Series、DataFrame 和 Panel。其中，Series 用于保存一维类的数据，DataFrame 用于保存二维类的数据，Panel 用于保存三维类或者可变维度的数据。在通常的数据分析中，我们经常使用 Series 和 DataFrame 这两种类型的数据，所以这两种类型要重点介绍。Panel 较少用到，但本着内容完整的原则，也做简要介绍。

首先引入已经安装的 Pandas。

```
In [1]: import numpy as np
        import pandas as pd
```

注意，习惯引入 Pandas 后更名为 pd，此处请从俗。

1．Series 对象

在 NumPy 中，我们已经知道如何创建一个一维数组了，并且该数组的每个元素都对应着从 0 开始编号的索引，比如 2016 年我国城市 GDP 前 8 名的数组，可以是下面这样的。

```
In [2]: g = np.array([27466.15, 24899.3, 19610.9, 19492.4, 17885.39, 17558.76, 15475.09,
                      12170.2])
```

要从这组数据中找出上海的 GDP 数值，可以用 g[0]，上海 GDP 排第一，比较醒目；要找苏州的 GDP 数值，一般人就不知道了，因为多数人看的是前三甲，虽然 g[6]就对应着苏州的

GDP 数值。从数据结构的角度来看，数组的索引缺乏明确的意义。

（1）创建 Series 对象

Pandas 中的 Series 对象解决了索引意义不明确的问题，如下所示，在原来数组 g 的基础上创建一个 Series 对象。

```
In [3]: gdp = pd.Series(g, index=['shanghai', 'beijing', 'guangzhou', 'shenzhen',
                                  'tianjin', 'chongqing', 'suzhou', 'chengdu'])
        gdp
Out[3]: shanghai     27466.15
        beijing      24899.30
        guangzhou    19610.90
        shenzhen     19492.40
        tianjin      17885.39
        chongqing    17558.76
        suzhou       15475.09
        chengdu      12170.20
        dtype: float64
```

In[3]中创建了 Series 对象，从 Out[3]的输出结果中可以看出，Series 对象有别于 NumPy 的 ndarray 对象，它所显示的内容不仅包括元素，也包括索引，并且索引还可以有意义（如图 2-1-1 所示）。

图 2-1-1　Series 对象的意义

创建 Series 对象就是建立 Series 类的实例，其完整的参数列表是：

```
pd.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)
```

- data：可以是数组、列表等类数组对象，也可以是字典，还可以是一个数字、字符串。In[2]中的 Series 对象就是基于数组而创建的，后面会演示 data 传入字典创建 Series 对象的方式。
- index：Series 对象的索引，如果为 None，则按照默认的从 0 开始的整数进行索引。注意 index 值的长度和 data 值的长度应该相同。为了将 index 所规定的索引在表述上和我们已经习惯的整数表示的索引区分开，将 index 的值所确定的索引称为“标签索引”，而在数组或者列表中用 0、1、2 等整数表示的元素排序位置的索引称为“位置索引”。当然，在本书的后续介绍中，有时候也笼统地使用“索引”。遇到此种情况，读者要根据上下文理解该处“索引”的意思。

还是使用数组对象 g 创建一个 Series 对象，这次 index 参数不传入值，结果与下述演示的结果一样。

```
In [4]: g2 = pd.Series(g)
        g2
Out[4]: 0    27466.15
        1    24899.30
        2    19610.90
        3    19492.40
        4    17885.39
        5    17558.76
        6    15475.09
        7    12170.20
        dtype: float64
```

如果 data 是列表，则效果与数组类似。

```
In [5]: pd.Series(data=[100, 200, 300])
Out[5]: 0    100
        1    200
        2    300
        dtype: int64
```

如果 data 是一个数字或者字符串，则会显示如下结果（也可以是一个字符串）。

```
In [6]: pd.Series(100, index=['a', 'b', 'c'])
Out[6]: a    100
        b    100
        c    100
        dtype: int64
```

在 pd.Series()的参数列表中，index 的值不仅可以是字符串组成的列表，还可以是数字。比如：

```
In [7]: d = pd.Series([0.1, 0.2, 0.3, 0.4], index=[1, 2, 3, 4])
        d
Out[7]: 1    0.1
        2    0.2
        3    0.3
        4    0.4
        dtype: float64
```

这样做的结果就是用 index 的值覆盖了原来的位置索引的值，因为二者都是整数，所以当访问 d[1]的时候，就不是按照位置索引返回第二个元素的值（0.2）了，而是根据 index 定义的索引返回第一个元素的值（0.1）。

```
In [8]: d[1]
Out[8]: 0.10000000000000001      #能解释这个结果吗
```

当 data 是字典的时候，与上述稍有不同，Series 对象的标签索引可以由字典的 key 来充任。

```
In [9]: gdp2 = pd.Series({"wuhan":11912.6, "hangzhou":11050.5, "nanjing":10503})
        gdp2
Out[9]: hangzhou    11050.5
        nanjing     10503.0
        wuhan       11912.6
        dtype: float64
```

字典本来没有顺序，但是索引是有顺序的。在 In[9]中使用字典创建 Series 对象时，Pandas 将字典的键排序后作为索引——请仔细观察 Out[9]的显示结果。那么，index 能不能在创建 Series

对象的时候就约定好呢？当然可以，不要忘记那个 index 参数。

```
In [10]: gdp3 = pd.Series({"wuhan":11912.6, "hangzhou":11050.5, "nanjing":10503},
                          index=['nanjing', 'wuhan', 'hangzhou'])
         gdp3
Out[10]: nanjing    10503.0
         wuhan      11912.6
         hangzhou   11050.5
         dtype: float64
```

（2）简单的 Series 对象属性

对于每个 Series 对象，都有 index 和 values 两个基本属性，可以分别获得标签索引和元素。

```
In [11]: gdp.index
Out[11]: Index(['shanghai', 'beijing', 'guangzhou', 'shenzhen', 'tianjin', 'chongqing',
                'suzhou', 'chengdu'],
               dtype='object')

In [12]: gdp.values
Out[12]: array([ 27466.15,  24899.3 ,  19610.9 ,  19492.4 ,  17885.39,  17558.76,
                 15475.09,  12170.2 ])
```

对于标签索引，不仅在创建 Series 对象的时候可以设置，还能够在以后进行修改。

```
In [13]: gdp.index = ["SHANGHAI", "BEIJING", "GUANGZHOU", "SHENZHEN", "TIANJIN",
                      "CHONGQING", "SUZHOU", "CHENGDU"]
```

如果再次查看 gdp，就会发现索引已经全变成大写字母拼写的城市名称了。对于 In[4]中所创建的 g2，也可以通过对 g2.index 赋值，实现有明确意义的标签索引。

Series 对象还有一个 name 属性，通过这个属性能够对当前对象进行描述。

```
In [14]: gdp.name = "GDP(hunder million RMB)"
         gdp
Out[14]: SHANGHAI     27466.15
         BEIJING      24899.30
         GUANGZHOU    19610.90
         SHENZHEN     19492.40
         TIANJIN      17885.39
         CHONGQING    17558.76
         SUZHOU       15475.09
         CHENGDU      12170.20
         Name: GDP(hunder million RMB), dtype: float64
```

其实，在创建 Series 对象的参数列表中就有 name 参数。如果传入的是对本对象的描述，则和上述操作效果相同。

此外，标签索引也有 name 属性，可以通过它为标签索引做注释，或者理解为标签索引的“名称”。

```
In [15]: gdp.index.name = "City Name"
         gdp
Out[15]: City Name
         SHANGHAI     27466.15
         BEIJING      24899.30
```

```
GUANGZHOU    19610.90
SHENZHEN     19492.40
TIANJIN      17885.39
CHONGQING    17558.76
SUZHOU       15475.09
CHENGDU      12170.20
Name: GDP(hunder million RMB), dtype: float64
```

在学习 Python 的时候，我们已经认识到“万物皆对象”。刚才对标签索引属性 name 的操作，以及前面用 gdp.index 查看标签索引，都意味着标签索引就是一个对象——Index 对象。如果使用 dir(gdp.index)查看这个对象，会发现它有很多属性和方法。

```
In [16]: type(gdp.index)
Out[16]: pandas.core.indexes.base.Index

In [17]: dir(gdp.index)    #省略输出内容，请读者自行查看。
```

索引对象是我们要在后面专门学习的。

这里姑且对 Series 对象有一些认识，下一节会深入讲解。下面创建 DataFrame 对象，它比 Series 对象多了一个维度，又跟 Series 对象“貌离神合”。

2．DataFrame 对象

在 Pandas 中，Series 对象用于存储一维数据，DataFrame 专门用于存储二维数据。二维数据很常见，特别是在网站开发中（推荐参考阅读《跟老齐学 Python：Django 实战》），要用到数据库。每个具体的数据保存到数据库中的表里面，那么每个表就是一个二维数据。

图 2-1-2 所展示的就是一个数据库表（来自《跟老齐学 Python：Django 实战》中的一个项目的数据库表）。仔细观察这个数据库表的样子，跟我们熟悉的电子表格差不多。是的，基本结构是一样的。下面再看一种常见的电子表格，如图 2-1-3 所示。

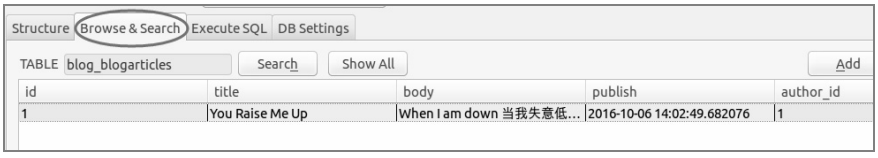


图 2-1-2　关系型数据库表

	A	B	C
1	City_Name	GDP	Population
2	SHANGHAI	27466.15	2419.70
3	BEIJING	24899.30	2172.90
4	GUANGZHOU	19610.90	1350.11
5	SHENZHEN	19492.60	1137.87
6	TIANJIN	17885.39	1562.12
7	CHONGQING	17558.76	3016.55
8	SUZHOU	15475.09	1375.00
9	CHENGDU	12170.20	1591.76

图 2-1-3　电子表格

如果把图 2-1-3 中的数据看作一个对象，是否可以创建某种数据结构呢？如果用数组，可以用二维数组，但是表中 A 列的索引和 1 行的记录就体现不出来了，当然，可以自定义一种新的数组类型满足这个要求，这仅仅是一种解决方法；而 Series 对象又是一维的。这时就要用新

的范式来解决了，于是 DataFrame 对象应运而生。

（1）基本方法

在 Pandas 中，通过实例化 pd.DataFrame()创建 DataFrame 对象。这种方法跟创建 Series 对象类似，而且参数列表中也有很多相似的。

```
pd.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)
```

这是创建 DataFrame 对象的基本方法。下面对参数列表中几个参数的含义进行说明和演示，帮助读者理解这种创建 DataFrame 对象的基本方法。

- data：可以是嵌套列表、二维数组、字典或者 DataFrame 对象。字典中的数据除字符串、数字等基本的 Python 数据类型外，还可以包括 Series、数组等序列类型的对象。
- index：可以是索引对象或者类数组对象。当然，跟 Series 类似，如果 index 为 None，则按照默认的 0、1、2……顺序建立索引。
- columns：可以是索引对象（后面会专门介绍索引对象）或者类数组对象。其含义是列索引。如果用数据库或者电子表格中的术语，则可以叫作“字段”。对照图 2-1-3，就是指“GDP”和“Population”。在创建 DataFrame 对象的时候，也可以为空，这时就用与“index 为空”一样的方式处理。

以上几个是重点，其他参数就不介绍了，读者能够理解。如果不清楚，特别建议查看官方文档。其实，笔者对参数的这些解释，也是根据自己对官方文档的理解而来的——显然掌握某种技术并不神秘，关键是要阅读文档。

下面就通过示例从不同侧面理解创建 DataFrame 对象的基本方法。

```
In [18]: gp = pd.DataFrame([[27466.15, 2419.70], [24899.30, 2172.90],
                            [19610.90, 1350.11], [19492.60, 1137.87],
                            [17885.39, 1562.12], [17558.76, 3016.55],
                            [15475.09, 1375.00], [12170.20, 1591.76]])
         gp
Out[18]:
               0       1
         0  27466.15  2419.70
         1  24899.30  2172.90
         2  19610.90  1350.11
         3  19492.60  1137.87
         4  17885.39  1562.12
         5  17558.76  3016.55
         6  15475.09  1375.00
         7  12170.20  1591.76
```

In[18]中创建了 DataFrame 对象，从 Out[18]输出结果中可以看出，它就是一个二维表格。在创建这个 DataFrame 对象的语句中，没有向 index 和 columns 两个参数传入值，就用从 0 开始计数的整数表示相应索引。与 Series 对象类似，也能够通过 DataFrame 对象的 index 和 columns 两个属性为其设置“标签索引”，如图 2-1-4 所示。

```
In [19]: gp.index = ['SHANGHAI', 'BEIJING', 'GUANGZHOU', 'SHENZHEN', 'TIANJIN',
                     'CHONGQING', 'SUZHOU', 'CHENGDU']
         gp.columns = ['GDP', 'Population']
```

```
In [19]: gp
Out[19]:
                      GDP     Population
SHANGHAI     27466.15     2419.70
BEIJING      24899.30     2172.90
GUANGZHOU    19610.90     1350.11
SHENZHEN     19492.60     1137.87
TIANJIN      17885.39     1562.12
CHONGQING    17558.76     3016.55
SUZHOU       15475.09     1375.00
CHENGDU      12170.20     1591.76
```

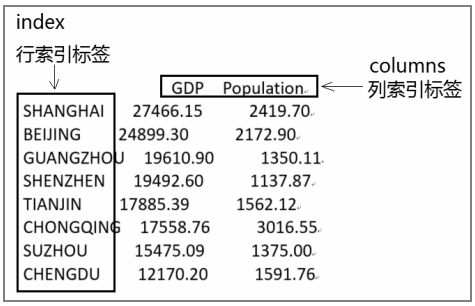


图 2-1-4　设置标签索引

如果在创建 DataFrame 对象的时候，给 index 和 columns 传入参数，则可以用如下方式。

```
In [20]: gp = pd.DataFrame([[27466.15, 2419.70], [24899.30, 2172.90], [19610.90, 1350.11],
                            [19492.60, 1137.87], [17885.39, 1562.12],
                            [17558.76, 3016.55], [15475.09, 1375.00],
                            [12170.20, 1591.76]],
                            index=['SHANGHAI', 'BEIJING', 'GUANGZHOU', 'SHENZHEN',
                                   'TIANJIN', 'CHONGQING', 'SUZHOU', 'CHENGDU'],
                            columns=['GDP', 'Population'])
```

还有与 Series 对象类似的操作，可以分别给 index 和 columns 命名，如下所示。

```
In [21]: gp.index.name = 'City_Name'
         gp.columns.name = "Items"
         gp
Out[21]:
Items            GDP     Population
City_Name
SHANGHAI     27466.15     2419.70
BEIJING      24899.30     2172.90
GUANGZHOU    19610.90     1350.11
SHENZHEN     19492.60     1137.87
TIANJIN      17885.39     1562.12
CHONGQING    17558.76     3016.55
SUZHOU       15475.09     1375.00
CHENGDU      12170.20     1591.76
```

data 参数所引用的值除上面的嵌套列表（该列表可以转换为二维数组）外，还可以是字典类型的数据。

```
In [22]: pd.DataFrame({"city":["beijing", "beijing", "hubei", "shanghai"],
                       "marks": [100.00, 96.91, 82.57, 82.47]},
                       index=["PKU", "Tsinghua", "WHU", "Fudan" ])
Out[22]:
                    city      marks
         PKU       beijing    100.00
         Tsinghua  beijing    96.91
         WHU       hubei      82.57
         Fudan     shanghai   82.47
```

从输出结果中不难理解 In[22]中创建 DataFrame 对象的各个参数的含义。字典的 key 充当了列的标签索引，字典的 value 是具体的数据，index 的值是行标签索引。

在 In[22]中，字典的 value 是一个列表，如果换成 Series 对象，也是可以的。建议读者自行检测。

用于创建 DataFrame 对象的数据，也包括自定义类型的数组，如下所示。

```
In [23]: u = np.array([("beijing", 100.00), ("beijing", 96.91), ("hubei", 82.57),
                       ("shanghai", 82.47)], dtype=[("city", "30S"), ("marks", np.float)])
         pd.DataFrame(u, index=["PKU", "Tsinghua", "WHU", "Fudan"])
Out[23]:
                      city        marks
         PKU        b'beijing'    100.00
         Tsinghua   b'beijing'    96.91
         WHU        b'hubei'      82.57
         Fudan      b'shanghai'   82.47
```

在数据分析项目中，除有可能用刚才演示的基本方法创建 DataFrame 对象外，还会遇到不同类型的数据之间的转换。

（2）数据格式转换

虽然使用 np.DataFrame()创建 DataFrame 对象的时候，其 data 是前述某种格式的数据，可以理解为将该类型数据转换为 DataFrame 对象，但那毕竟还是作为 DataFrame 类实例化时所需要的参数。为了清晰，将其归类为创建 DataFrame 对象的“基本方法”。这里要讲述的“数据格式转换”则是使用一些命名上有明显标识的转换方法。

在 Jupyter 中，输入“pd.DataFrame.from_”，接着按 Tab 键，会看到如图 2-1-5 所示的效果。

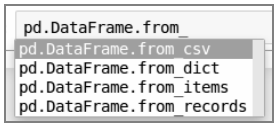


图 2-1-5　pd.DataFrame()类中以 from 开头的方法

图 2-1-5 显示了 pd.DataFrame()类中以 from 开头的方法——类似的查看方法前面讲过了。从名称上就可以断定，这些方法的作用是将某种格式的数据转换为 DataFrame 对象。下面举几个例子，其他的方法请读者自行探究。

先看如何把字典数据转换为 DataFrame 对象，当然，用前面的“基本方法”也可以，不过这里非要用 pd.DataFrame.from_dict()不可。

```
pd.DataFrame.from_dict(data, orient='columns', dtype=None)
```

从官方文档的解释中可以获悉，上面所示方法中参数 data 就是字典，而字典的格式可以是{field : array-like}或者{field : dict}样式。另外一个参数 orient 可以有两个值，默认是 columns，意思是字典的 key 是列的标签索引；如果 orient='index'，则意味着字典的 key 作为行的标签索引。

```
In [24]: dict_gdp = {"GDP": [27466.15, 24899.30, 19610.90, 19492.60],
                     "Population": [2419.70, 2172.90, 1350.11, 1137.87]}
                     #{field : array-like}类型
         pd.DataFrame.from_dict(dict_gdp)    #默认 orient="columns"
Out[24]:
               GDP     Population
         0  27466.15     2419.70
         1  24899.30     2172.90
         2  19610.90     1350.11
         3  19492.60     1137.87

In [25]: pd.DataFrame.from_dict(dict_gdp, orient="index")
Out[25]:
                            0        1         2         3
              GDP    27466.15  24899.3  19610.90  19492.60
        Population   2419.70   2172.9   1350.11   1137.87
```

In[24]中创建的字典是{field : array-like}样式的，下面再创建一个{field : dict}样式的，看看转换结果。

```
In [26]: dict_gdp2 = {"GDP": {"SHANGHAI":27466.15, "BEIJING":24899.30,
                              "GUANGZHOU":19610.90, "SHENZHEN":19492.60},
                      "Population": {"SHANGHAI":2419.70, "BEIJING":2172.90,
                                     "GUANGZHOU":1350.11, "SHENZHEN":1137.87}}
         pd.DataFrame.from_dict(dict_gdp2)
Out[26]:
                         GDP     Population
         BEIJING      24899.30     2172.90
         GUANGZHOU    19610.90     1350.11
         SHANGHAI     27466.15     2419.70
         SHENZHEN     19492.60     1137.87

In [27]: pd.DataFrame.from_dict(dict_gdp2, orient="index")
Out[27]:
                     GUANGZHOU   SHANGHAI   BEIJING  SHENZHEN
              GDP     19610.90   27466.15   24899.3  19492.60
        Population    1350.11    2419.70    2172.9   1137.87
```

为了理解 pd.DataFrame.from_dict()方法的应用，读者一定要将上述操作认真对比。笔者的解释是苍白的，唯有读者通过对比领悟才是深刻的。

还记得字典对象有 items()方法吗？在前面看到的 pd.DataFrame()类中有一个名为 from_items()的方法，是不是跟字典的 items()方法有关系呢？揭开这个奥秘的方法读者早已熟知，也是笔者反复强调的，这就是“渔”，比“鱼”更重要。

```
In [28]: pd.DataFrame.from_items?
Signature: pd.DataFrame.from_items(items, columns=None, orient='columns')
```

```
Docstring:
Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the
columns, but depends on the specified orientation). The values should be arrays or Series.
……    #省略显示后续内容
```

当读者阅读完文档之后，再看下面的例子，就豁然开朗了。

```
In [29]: items_gdp = dict_gdp.items()
         items_gdp
Out[29]: dict_items([('Population', [2419.7, 2172.9, 1350.11, 1137.87]),
                     ('GDP', [27466.15, 24899.3, 19610.9, 19492.6])])

In [30]: pd.DataFrame.from_items(items_gdp,
                                 columns=["shanghai", "beijing", "guangzhou",
                                 "shenzhen"],orient="index")
Out[30]:
                    shanghai  beijing  guangzhou  shenzhen
             GDP     27466.15  24899.3   19610.90  19492.60
         Population   2419.70   2172.9    1350.11   1137.87
```

在 In[30]的操作中，把几个参数都用上了，请读者根据已有经验和对文档的理解，猜测该操作的用意，并对照返回结果检查自己的猜测是否正确。

pd.DataFrame.from_*类的方法是把其他类型的数据转换为 DataFrame 对象，那么，其逆过程是否有函数呢？必须有，如图 2-1-6 所示。

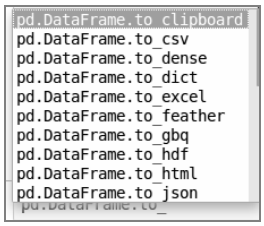


图 2-1-6　pd.DataFrame()类中以 to 开头的方法

图 2-1-6 是怎么出来的？读者应该知道。如果不知道，说明阅读本书不仔细。

至于图 2-1-6 中所示的这些方法如何使用，读者应该已经掌握了学习方法，所以这里不详细解释。如果觉得没有掌握学习方法，请将前述 pd.DataFrame.from_dict()的相关内容再读一遍。下面举个例子。

```
In [31]: gp
Out[31]:
         Items          GDP      Population
         City_Name
         SHANGHAI     27466.15     2419.70
         BEIJING      24899.30     2172.90
         GUANGZHOU    19610.90     1350.11
         SHENZHEN     19492.60     1137.87
         TIANJIN      17885.39     1562.12
         CHONGQING    17558.76     3016.55
         SUZHOU       15475.09     1375.00
         CHENGDU      12170.20     1591.76
```

```
In [32]: gp.to_csv("/home/qiwsir/Documents/data_analysis/gp.csv",
                   columns=["GDP", "Population"],
                   index_label=["SHANGHAI", "BEIJING", "GUANGZHOU", "SHENZHEN",
                                "TIANJIN", "CHONGQING", "SUZHOU", "CHENGDU"],
                   header=False)
In [33]: !head / home/qiwsir/Documents/DataAnalysis/chapter02/gp.csv
         SHANGHAI,27466.15,2419.7
         BEIJING,24899.3,2172.9
         GUANGZHOU,19610.9,1350.11
         SHENZHEN,19492.6,1137.87
         TIANJIN,17885.39,1562.12
         CHONGQING,17558.76,3016.55
         SUZHOU,15475.09,1375.0
         CHENGDU,12170.2,1591.76
```

gp 是我们在前面已经建立的一个 DataFrame 对象，gp.to_csv()将这个 DataFrame 对象保存到.csv 文件中。请读者注意查看文档，当输入“pd.DataFrame.to_csv?”的时候，会出现如下文字显示：

```
Signature: pd.DataFrame.to_csv(self, path_or_buf=None,……)    #参数很多，省略显示
```

这里重点说明这行文字中的一个细节。

读者一定没有忘记类的方法中第一参数必须是 self 吧（忘记也没关系，赶快找到《跟老齐学 Python：轻松入门》看一下），self 代表什么呢？实例！对，是实例。在创建 DataFrame 的基本方法中已经知道，gp 就是 pd.DataFrame 类对象的实例，所以还可以像下面这样操作。

```
In [34]: pd.DataFrame.to_csv(gp, "/home/qiwsir/Documents/data_analysis/gp2.csv",
                              columns=["GDP", "Population"],
                              index_label=["SHANGHAI", "BEIJING", "GUANGZHOU",
                                           "SHENZHEN","TIANJIN", "CHONGQING",
                              "SUZHOU", "CHENGDU"], header=False)

In [35]: ! !head /home/qiwsir/Documents/DataAnalysis/chapter02/gp2.csv
         SHANGHAI,27466.15,2419.7
         BEIJING,24899.3,2172.9
         GUANGZHOU,19610.9,1350.11
         SHENZHEN,19492.6,1137.87
         TIANJIN,17885.39,1562.12
         CHONGQING,17558.76,3016.55
         SUZHOU,15475.09,1375.0
         CHENGDU,12170.2,1591.76
```

In[32]和 In[34]虽然表面上看不一样，但其本质相同。读者若不理解，请参考前述推荐阅读的那本书。

用刚才演示的方式将数据写入一个.csv 文件中，那么反过来，从.csv 文件中读取数据，也应该是理所应当的了。不过，我们把文件的“读取和写入”放在后面的 2.4 节单独讲述。

3．Panel 对象

Pandas 中的三种类型的数据各司其职，Series 对象存储一维数据，DataFrame 对象存储二维

数据，Panel 对象存储三维数据，那么更高维度呢？一来在实践中很少用，二来本质上可以用前述三种数据表示。甚至三维数据，也能够用 Series 和 DataFrame 表示。不过，Pandas 还是提供了建构三维数据的对象类型，让我们的操作更便捷了。

首先要知道三维数据是什么样子的。下面还是用二维方式展示一下三维数据，如图 2-1-7 所示。

Nation	City	Average Temperature(℃)	Altitude(m)
China	Beijing	11.8	37
	HongKong	22.9	33
USA	Chicago	9.5	186
	New York	12.6	3
	San Fran	13.9	16

图 2-1-7　二维方式展示的三维数据

如果把 Nation 看作一个维度，那么上面的数据就可以看作是一个三维数据，再转换一下，读者可能就更清楚了。

把表转换为图 2-1-8 所示的坐标形式，从中可以看出，表中的任何一个数据都可以对应着三维坐标空间中的一个点。这个点是由三个坐标轴共同确定的，或者说我们已经从数据表中提炼出来三个轴。

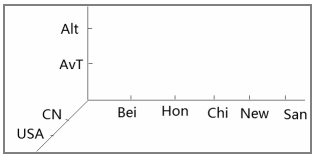


图 2-1-8　坐标形式展示的三维数据

再来看 Pandas 中创建 Panel 对象的方法，也就是 pd.Panel()这个类的详情。

```
pd.Panel(data=None, items=None, major_axis=None, minor_axis=None, copy=False,
        dtype=None)
```

下面对照前面提炼出来的三维坐标轴和数据表，分别解释 items、major_axis、minor_axis 的含义。

- items：即 axis=0，在上面的三维数据示例中对应着“CN”和“USA”。再观察每一项所对应的值，都可以看作是一个 DataFrame 对象。比如“CN”所对应的是图 2-1-9 所示的二维数据。

City	Average Temperature(℃)	Altitude(m)
Beijing	11.8	37
HongKong	22.9	33

图 2-1-9　二维数据

- major_axis：即 axis=1，在上面三维数据示例中对应着“Beijing”、“HongKong”等城市名称的索引，其实就是 DataFrame 对象的标签索引。
- minor_axis：即 axis=2，在上面的三维数据示例中对应着“Averge Temperature”和“Altitude”列标签索引。

而 data 所允许的数据类型就比较多了，比如三维数组、嵌套列表和字典等。

```
In [36]: a = np.random.rand(3, 4, 5)
         p = pd.Panel(a)
         p
Out[36]: <class 'pandas.core.panel.Panel'>
         Dimensions: 3 (items) x 4 (major_axis) x 5 (minor_axis)
         Items axis: 0 to 2
         Major_axis axis: 0 to 3
         Minor_axis axis: 0 to 4

In [37]: a.shape
Out[37]: (3, 4, 5)
```

数组 a 是一个三维数组，在 In[35]中以这个数组为 data 创建了 Panel 对象 p，从 p 的输出中可以看到该 Panel 对象的三个轴及其索引范围，正好跟数组三个轴的索引范围对应。

如果用前面表格所示数据创建一个 Panel 对象，则参数列表中的 data 可以引用一个字典类型对象，并且该字典的 value 使用 DataFrame 类型的数据。

```
In [38]: cdf = pd.DataFrame([{"Average Temperature":11.8, "Altitude":37},
                             {"Average Temperature":22.9, "Altitude":33}],
                             index=["Beijing", "HongKong"])
         udf = pd.DataFrame([{"Average Temperature":9.5, "Altitude":186},
                             {"Average Temperature":12.6, "Altitude":3},
                             {"Average Temperature":13.9, "Altitude":16}],
                             index=["Chicago", "New York", "San Fran"])
         ncaa = pd.Panel({"China":cdf, "USA":udf})
         ncaa
Out[38]: <class 'pandas.core.panel.Panel'>
         Dimensions: 2 (items) x 5 (major_axis) x 2 (minor_axis)
         Items axis: China to USA
         Major_axis axis: Beijing to San Fran
         Minor_axis axis: Altitude to Average Temperature
```

在创建 Panel 对象的时候，可以声明 items、major_axis、minor_axis 三个参数的值，从而确定各个轴的标签索引，而不是使用默认的整数。

```
In [39]: p = pd.Panel(data=a, items=["itema", "itemb", "itemc"],
                      major_axis=['a', 'b', 'c', 'd'],
                      minor_axis=['one', 'two', 'three', 'four', 'five'])    #a 是前面
         所创建的三维数组
         p
Out[39]: <class 'pandas.core.panel.Panel'>
         Dimensions: 3 (items) x 4 (major_axis) x 5 (minor_axis)
         Items axis: itema to itemc
         Major_axis axis: a to d
         Minor_axis axis: one to five
```

接下来学习根据各轴的名称读取 Panel 中部分数据的方法，返回的都是 DataFrame 对象。

```
In [40]: p['itema']
Out[40]:
              one       two     three      four      five
        a  0.075884  0.856519  0.747313  0.519158  0.551971
        b  0.872562  0.419473  0.320454  0.209069  0.527021
```

```
        c  0.632932  0.390712  0.740557  0.229337  0.896173
        d  0.041699  0.817207  0.500096  0.903874  0.333724

In [41]: p['itemb']
Out[41]:
               one       two     three      four      five
        a  0.478257  0.133654  0.953475  0.832498  0.367102
        b  0.503323  0.863795  0.357759  0.868674  0.612014
        c  0.522115  0.865674  0.881541  0.702868  0.450352
        d  0.168332  0.426337  0.107180  0.738528  0.206686

In [42]: p.major_xs('a')
Out[42]:
                 itema     itemb     itemc
        one    0.075884  0.478257  0.522365
        two    0.856519  0.133654  0.089097
        three   0.747313  0.953475  0.936472
        four    0.519158  0.832498  0.403946
        five    0.551971  0.367102  0.739752

In [43]: p.minor_xs('one')
Out[43]:
             itema     itemb     itemc
        a  0.075884  0.478257  0.522365
        b  0.872562  0.503323  0.044867
        c  0.632932  0.522115  0.900281
        d  0.041699  0.168332  0.059532
```

在数据分析实践和数据分析相关的书籍资料中，都很少涉及 Panel 对象，因为事实上类似前面标有国家、城市的温度和海拔的数据表，可以利用 DataFrame 对象的多级索引，即 MultiIndex 对象解决——多级索引也是后面要详述的。

2.2 索引对象

Pandas 相对 NumPy 的一大变化就是丰富了数据的索引及其方法。

在 Pandas 中，索引是对象，不仅可以有一级，还可以有多级。也正是因为多级索引的存在，才能够将本来多维的数据结构转换为二维或者一维，即 DataFrame 对象或者 Series 对象。所以，从本节开始，我们的焦点就放在 Series 和 DataFrame 这两种对象上了。首先要研究的就是它们的索引——Index 对象和 MultiIndex 对象。

1．Index 对象

前面我们已经遇到索引对象 Index 了，例如每个 Series 对象都有一个 index 属性，通过此属性得到的就是 Index 对象。

```
In [1]: import numpy as np
        import pandas as pd
        s = pd.Series(np.random.randn(5))
        type(s.index)
Out[1]: pandas.core.indexes.range.RangeIndex
```

```
In [2]: s.index
Out[2]: RangeIndex(start=0, stop=5, step=1)
```

从 Out[2]的结果不难断定，Index 对象也是序列，这个序列兼具了数组的一些特征，可以看成是一维数组。

```
In [3]: s.index[0]
Out[3]: 0

In [4]: s.index[:3]
Out[4]: RangeIndex(start=0, stop=3, step=1)

In [5]: s.index[s.index>3]
Out[5]: Int64Index([4], dtype='int64')
```

但是跟一维数组最大的区别在于，Index 对象是不可变的，这种“不可变”特性使得它能够被不同的 Series 对象或者 DataFrame 对象使用，并且实现后面会提到的“自动对齐”功能。

```
In [6]: s.index[0] = "a"
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-428-95fbfde5048a> in <module>()
----> 1 s.index[0] = "a"

/usr/local/lib/python3.5/dist-packages/pandas/core/indexes/base.py   in   __setitem__
(self, key, value)
   1668
   1669     def __setitem__(self, key, value):
-> 1670         raise TypeError("Index does not support mutable operations")
   1671
   1672     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

在创建 Series 对象的时候，我们已经通过 index 属性修改其“标签索引”的内容。

```
In [7]: s.index = ["a", "b", "c", "d", "e"]
        s
Out[7]: a    0.973760
        b   -0.060068
        c    0.315142
        d    0.457990
        e    0.349823
        dtype: float64
```

到现在还没有新意呢，都是复习。已有经验表明，Python 中的对象都是通过类创建的，那么 Index 对象能不能通过类创建呢？

```
pd.Index(data=None, dtype=None, copy=False, name=None, fastpath=False,
         tupleize_cols=True, **kwargs)
```

Index 对象就是 pd.Index()的实例。

```
In [8]: ind = pd.Index(["physics", "python", "math", "english"])    #①
```

```
        s = pd.Series([100, 90, 80, 70], index=ind)    #②
        s
Out[8]: physics  100
        python   90
        math     80
        english  70
        dtype: int64

In [9]: d = pd.DataFrame({"soochow":s, "tsinghua":[98, 34, 83, 49]}, index=ind)
        d
Out[9]:
               soochow   tsinghua
       physics   100       98
       python     90       34
       math       80       83
       english    70       49
```

In[8]的①创建了一个 Index()实例对象，并且把它应用到 In[8]的②和 In[9]中，分别创建 Series 对象和 DataFrame 对象。

关于 Index 对象，除可以看作是一维数组（像操作一维数组那样操作它）外，还可以看作是集合（可以对其使用集合的关系运算，这也是集合的特点）。

```
In [10]: inda = pd.Index([2,4,6,8])
         indb = pd.Index([3,4,7,8])

In [11]: inda | indb
Out[11]: Int64Index([2, 3, 4, 6, 7, 8], dtype='int64')

In [12]: inda & indb
Out[12]: Int64Index([4, 8], dtype='int64')

In [13]: inda ^ indb
Out[13]: Int64Index([2, 3, 6, 7], dtype='int64')
```

Index 对象本身还有很多其他的方法，在项目实践中如果读者遇到了，可以使用笔者在本书中一再演示并提及的方法去学习。

2．MultiIndex 对象

回顾曾经用过的一个数据表，如图 2-2-1 所示。

Nation	City	Average Temperature(℃)	Altitude(m)
China	Beijing	11.8	37
	HongKong	22.9	33
USA	Chicago	9.5	186
	New York	12.6	3
	San Fran	13.9	16

图 2-2-1　二维方式展示的三维数据

如果不用 Panel 对象，而是从二维数据表的角度来看，Nation 和 City 两列就是两层索引，按照 Python 的习惯，可以将 Nation 列称为第 0 级索引，将 City 列称为第 1 级索引。从这个例子中，读者肯定能够得到启发，类似这样的多级索引的表格在社会生活、生产实践中还有很多。

所以，我们需要学习 MultiIndex（多级索引）。

Pandas 里面提供了多种方法用于创建 MultiIndex 对象，以及在 Series 对象和 DataFrame 对象中应用它。

（1）通过函数创建

在 Jupyter 中输入 dir(pd)，在结果中查看一番，有没有名为 MultiIndex 的类？这纯粹是一种探索。如果有，则说明 Pandas 专门提供了创建 MultiIndex 对象的方法；否则，可能要另寻他路。

果然有！Pandas 就是这么“贴心”。

根据经验，类都有一些方法，所以还是输入“dir(pd.MultiIndex)”看一下 MultiIndex()类的属性和方法，发现有些方法是以 from 开头的——回忆前面的内容，也曾经遇到过。

在交互模式中输入“pd.MultiIndex.from_”，然后按 Tab 键，就能看到几个方法，它们都可以用来创建 MultiIndex 对象。

接下来就是查看文档了。

```
In [14]: pd.MultiIndex.from_tuples?
Signature: pd.MultiIndex.from_tuples(tuples, sortorder=None, names=None)
Docstring:
Convert list of tuples to MultiIndex

Parameters
----------
tuples : list / sequence of tuple-likes
    Each tuple is the index of one row/column.
sortorder : int or None
    Level of sortedness (must be lexicographically sorted by that
    level)

Returns
-------
index : MultiIndex
```

因为文档较短，所以全部复制了。首先，看文档的最后，这个方法返回的是 MultiIndex 对象；然后看参数，也比较简单，tuples 接收一个用列表或者类数组方式表示的多级索引。

```
In [15]: cities_index = [("China", "Beijing"), ("China", "HongKong"),
                         ("USA", "Chicago"), ("USA", "NewYork"), ("USA", "SanFran")]
         cities = pd.MultiIndex.from_tuples(cities_index)
         cities
Out[15]: MultiIndex(levels=[['China', 'USA'], ['Beijing', 'Chicago', 'HongKong',
                            'NewYork', 'SanFran']],
                     labels=[[0, 0, 1, 1, 1], [0, 2, 1, 3, 4]])
```

依照 pd.MultiIndex.from_tuples()方法的文档，我们创建了一个 MultiIndex 对象（In[15]中变量 cities 所引用的对象）。请注意观察 Out[15]的返回结果，其中包括 levels 和 labels 两个变量，在后面我们会用到它们。

上面讲解了 pd.MultiIndex.from_tuples()方法。对于另外两个方法，这里以简单示例说明，强烈建议读者看一下帮助文档。

```
In [16]: city_arr = np.array([["China", "China", "USA", "USA", "USA"],
                              ["Beijing", "HongKong", "Chicago", "NewYork", "SanFran"]])
         cities2 = pd.MultiIndex.from_arrays(city_arr)
         cities2
Out[16]: MultiIndex(levels=[['China', 'USA'], ['Beijing', 'Chicago', 'HongKong',
                            'NewYork', 'SanFran']],
                    labels=[[0, 0, 1, 1, 1], [0, 2, 1, 3, 4]])
```

使用 pd.MultiIndex.from_arrays()的时候，所传入的参数值也可以是类数组的其他序列，比如嵌套列表（[["China", "China", "USA", "USA", "USA"], ["Beijing", "HongKong", "Chicago", "NewYork", "SanFran"]]）。

另外要说明的就是 pd.MultiIndex.form_product()方法，它依据笛卡儿积的计算方法生成 MultiIndex 对象。

```
In [17]: pd.MultiIndex.from_product([['a', 'b'], [100, 200]])
Out[17]: MultiIndex(levels=[['a', 'b'], [100, 200]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

根据笛卡儿积的运算规则，['a', 'b']和[100, 200]相乘的结果应该是[['a', 100], ['a', 200], ['b', 100] ['b', 200]]，所以，In[17]与下述方法是等效的。

```
In [18]: pd.MultiIndex.from_arrays([['a', 'a', 'b','b'], [100, 200, 100, 200]])
Out[18]: MultiIndex(levels=[['a', 'b'], [100, 200]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

（2）通过实例化类创建

既然 pd.MultiIndex 是类对象，那么也就可以仿照 Index 对象创建方法那样，通过实例化 pd.MultiIndex()类来创建对象。先看一下 pd.MultiIndex()类的参数列表：

```
pd.MultiIndex(levels=None,labels=None,sortorder=None,names=None,copy=False,verify_in
tegrity=True, _set_identity=True, name=None, **kwargs)
```

要通过实例化 pd.MultiIndex()类创建一个不空的 MultiIndex 对象，就必须给 levels 和 labels 两个参数传入值。在前面创建的 MultiIndex 对象中已经看到了这两个参数，对照输出结果就能揣摩出这两个参数的含义。

- levels：以序列类数据表示标签索引。
- labels：以整数的方式表示每个标签索引的位置。

```
In [19]: pd.MultiIndex(levels=[['a', 'b'], [100, 200]], labels=[[0, 0, 1, 1], [0, 1, 0,
                       1]])
Out[19]: MultiIndex(levels=[['a', 'b'], [100, 200]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

In[19]和 In[17]操作等效。levels=[['a', 'b'], [100, 200]]表示在即将创建的多级索引中有两级索引，第 0 级的标签索引使用'a'和'b'，第 1 级的标签索引使用 100 和 200；labels=[[0, 0, 1, 1], [0, 1, 0, 1]]则规定了各级标签索引的位置，[0, 0, 1, 1]所规定的是第 0 级中'a'和'b'的位置，根据['a', 'b']可以知道，其标签索引的最终排列应该是['a', 'a', 'b', 'b']，后面的第 1 级与此雷同。

如此，通过实例化 pd.MultiIndex()类创建了 MultiIndex 对象。这仅是一种实例化的方法，我们还可以通过实例化 pd.Index()类创建 MultiIndex 对象。

```
In [20]: pd.Index([("a", 100), ("a", 200), ("b", 100), ("b", 200)])
```

```
Out[20]: MultiIndex(levels=[['a', 'b'], [100, 200]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

创建索引对象之后，就可以在创建 Series 对象或者 DataFrame 对象时通过 index 参数使用它了。

不论是 Index 索引对象还是 MultiIndex 索引对象，都有同一个重要的性质，那就是“不可变”，请读者谨记。

3．在数据中使用 MultiIndex 对象

在具体的数据对象中使用 Index 对象比较简单，前面已经有很多示例。下面仅就在具体数据中使用 MultiIndex 对象进行说明。

还是从简单的 Series 对象开始讲起。

```
In [21]: gdp_index = [("shanghai", 2015), ("shanghai", 2016), ("beijing", 2015),
                      ("beijing", 2016), ("guangzhou", 2015), ("guangzhou", 2016)]
         gdp_mind = pd.MultiIndex.from_tuples(gdp_index)
         gdp3 = pd.Series([25300, 27466, 23000, 24899, 18100, 19611], index=gdp_mind)
         gdp3
Out[21]: shanghai   2015    25300
                    2016    27466
         beijing    2015    23000
                    2016    24899
         guangzhou  2015    18100
                    2016    19611
         dtype: int64
```

看到这里，是不是总觉得 MultiIndex 在 gdp3 里是多余的呀？Out[21]的结果完全可以用前面已经学习过的 DataFrame 对象表示，不需要多级索引，如下所示。

```
            2015    2016
beijing    23000   24899
guangzhou  18100   19611
shanghai   25300   27466
```

这样是不是干净清爽了？

恭喜读者，你已经对 Pandas 的数据对象有了深刻的认识。这正说明高维度的数据都是由低维度的数据组成的。

在 Pandas 中，还有专门实现上述转换的函数，可以实现 Series 类型的对象和 DataFrame 类型的对象之间的互换。

```
In [22]: gdp3.unstack()
Out[22]:
                     2015    2016
         beijing    23000   24899
         guangzhou  18100   19611
         shanghai   25300   27466
```

对于 gdp3.unstack()的逆过程，也有！

```
In [23]: gdp3.unstack().stack()
Out[23]: beijing    2015    23000
                    2016    24899
```

```
         guangzhou   2015    18100
                     2016    19611
         shanghai    2015    25300
                     2016    27466
         dtype: int64
```

刚才提到，高维度的对象可以由低维度的对象组成。对此，在创建 DataFrame 对象的时候，读者应该已经体会到了。为了加深印象，再体会一次也无妨。

```
In [24]: price = [32260, 31670, 30972, 32131, 16153, 18484]
         gdp_house = pd.DataFrame({"GDP": gdp3, "Price": price})
         gdp_house
Out[24]:
                           GDP   Price
         shanghai  2015  25300  32260
                   2016  27466  31670
         beijing   2015  23000  30972
                   2016  24899  32131
         guangzhou 2015  18100  16153
                   2016  19611  18484
```

不过，这里不是简单的重复，从上述操作中可以看到，如果 Series 对象的索引是多级索引，并用它作为 DataFrame()类的字典参数中的一个 value，那么另外的 value 会根据 Series 对象的多级索引自动对齐，并且最终的 DataFrame 对象也使用 Series 对象的 MultiIndex 对象。

```
In [25]: gdp_house.index
Out[25]: MultiIndex(levels=[['beijing', 'guangzhou', 'shanghai'], [2015, 2016]],
                    labels=[[2, 2, 0, 0, 1, 1], [0, 1, 0, 1, 0, 1]])
```

就在刚才，我们已经在不知不觉中创建了一个 DataFrame 对象，并且其索引是 MultiIndex。针对这个 DataFrame 对象，还可用下面的方式创建。

```
In [26]: gdp4 = pd.Series({("shanghai", 2015): 25300, ("shanghai", 2016): 27466,
                           ("beijing", 2015): 23000, ("beijing", 2016): 24899,
                           ("guangzhou", 2015):18100, ("guangzhou", 2016):19611})
         gdp4
Out[26]: beijing    2015    23000
                    2016    24899
         guangzhou  2015    18100
                    2016    19611
         shanghai   2015    25300
                    2016    27466
         dtype: int64
```

这种创建 Series 对象的方法并不是新的，具体解释请参考前面的有关内容。不过与以往有区别的是字典的 key，key 作为 Series 对象的索引，它不是单一元素的数据（比如字符串、数字），而是由两个元素组成的元组，这时候 Pandas 会自动将两个元素处理为两级的索引。

当然，不通过 pd.Series()类，而是直接使用 pd.DataFrame()类也同样能够创建多级索引的数据，只是需要在实例化的时候传入 index 和 columns 的值。

```
In [27]: a = np.array([[25300, 32260], [27466, 31670], [23000, 30972],
                       [24899, 32131], [18100, 16153], [19611, 18484]])
         gdp_house2 = pd.DataFrame(a,
```

```
                                    index=[["shanghai", "shanghai", "beijing", "beijing",
                                            "guangzhou", "guangzhou"],
                                           [2015, 2016, 2015, 2016, 2015, 2016]],
                                    columns = ["GDP", "Price"])
        gdp_house2
Out[27]:
                           GDP   Price
        shanghai   2015  25300  32260
                   2016  27466  31670
        beijing    2015  23000  30972
                   2016  24899  32131
        guangzhou  2015  18100  16153
                   2016  19611  18484
```

特别注意观察 In[27]中传入 index 的值的特点——其实是用嵌套列表表示了多级索引，而后 Pandas 对嵌套列表中的多级索引进行自动处理，最终得到 Out[27]的结果。当然，不用这种嵌套列表，直接传入 MultiIndex 对象也是可以的。

```
In [28]: gh_mind = pd.MultiIndex.from_arrays([["shanghai", "shanghai", "beijing",
                                               "beijing", "guangzhou", "guangzhou"],
                                              [2015, 2016, 2015, 2016, 2015, 2016]])
         pd.DataFrame(a, index=gh_mind, columns=['GDP', 'Price'])
Out[28]:
                           GDP   Price
        shanghai   2015  25300  32260
                   2016  27466  31670
        beijing    2015  23000  30972
                   2016  24899  32131
        guangzhou  2015  18100  16153
                   2016  19611  18484
```

以 Index 对象或者 MultiIndex 对象为索引而创建的 Pandas 数据，不仅能够以 Series 和 DataFrame 类型表示更多维度的数据，还为数据的有关操作提供了新的、含义更明确的方式，比如 2.3 节中的切片，可以看到比数组中的切片更丰富的内容。

2.3　数据索引和切片

在 NumPy 的数组中，曾经介绍过索引和切片。根据艾宾浩斯遗忘曲线的规律，读者现在对那部分知识的印象应该比较模糊了，所以建议返回第 1 章，再复习一番，然后学习本节 Pandas 的 Series 和 DataFrame 对象的索引和切片。

通常而言，学习过程需要重复和大量练习，把知识内化为自己的技能，才能在使用的时候胸有成竹。

1．Series 对象

下面还是从 Series 对象开始讲解，因为它是 Pandas 中最简单的。

继续使用在 2.1 节中所创建的并由变量 gdp 引用的 Series 对象。

```
In [1]: import numpy as np
        import pandas as pd
```

```
In [2]: g = np.array([27466.15, 24899.3, 19610.9, 19492.4, 17885.39, 17558.76, 15475.09,
                      12170.2])
        gdp = pd.Series(g, index=['shanghai', 'beijing', 'guangzhou', 'shenzhen',
                                  'tianjin', 'chongqing', 'suzhou', 'chengdu'])
```

从 Python 中的序列类型对象到 NumPy 的数组，要获取原数据的一部分，我们都使用[ ]符号，现在对于 Pandas 的数据，也依然秉承这个做法。

```
In [3]: gdp['suzhou']
Out[3]: 15475.09
```

在[ ]里面放置的是标签索引的一个值，类似于数组中用整数作为下标。gdp['suzhou']以某一个标签索引值为下标，得到其对应的数据。

根据上面的操作，我们不难看出，在 Series 对象中，每个标签索引的值与数据值是一一对应的。索引与数据之间的这种映射关系，类似 Python 中字典对象的“键（key）—值（value）”之间的映射关系。所以，从这个侧面看，Series 对象是类字典的对象，那么也就可以使用字典的一些方法进行操作。

```
In [4]: "shanghai" in gdp
Out[4]: True

In [5]: "hangzhou" in gdp
Out[5]: False

In [6]: gdp.keys()
Out[6]: Index(['shanghai', 'beijing', 'guangzhou', 'shenzhen', 'tianjin', 'chongqing',
               'suzhou', 'chengdu'],
              dtype='object')

In [7]: list(gdp.items())
Out[7]: [('shanghai', 27466.150000000001),
         ('beijing', 24899.299999999999),
         ('guangzhou', 19610.900000000001),
         ('shenzhen', 19492.400000000001),
         ('tianjin', 17885.389999999999),
         ('chongqing', 17558.759999999998),
         ('suzhou', 15475.09),
         ('chengdu', 12170.200000000001)]

In [8]: gdp["hangzhou"] = 11050.5    #增加一项
        gdp
Out[8]: shanghai     27466.15
        beijing      24899.30
        guangzhou    19610.90
        shenzhen     19492.40
        tianjin      17885.39
        chongqing    17558.76
        suzhou       15475.09
        chengdu      12170.20
        hangzhou     11050.50
        dtype: float64
```

在具有映射关系的对象中，通常还支持用“.”符号访问某个值，比如：

```
In [9]: gdp.suzhou
Out[9]: 15475.09
```

但是，在这里笔者很明确地表态，不提倡这种方式，虽然读者在很多资料里都会看到如此使用。这是因为我们在 Python 中已经有了一个约定，类似这种模式所访问的是某个对象的属性或者方法（对象）。对于部分没有经过严格训练的数据工程师来说，或许认为这无所谓，但作为一个有点强迫症的“码农”，笔者是坚决反对的，虽然到目前为止 Pandas 还允许 In[9]的操作，并且也没有看到会取消这种操作的意向。以上是个人经验，仅供参考，不是必须遵守的。

一般来讲，经验都是从惨痛教训中总结出来的，有时候还是有参考价值的，比如下面的示例。

```
In [10]: s = pd.Series(np.random.randn(4), index=["tot", "pop", "sos", "mom"])
         s.tot
Out[10]: -0.3062177474165233    #因为是随机值，所以读者看到的显示结果可能与此不同

In [11]: s.pop
Out[11]: <bound method NDFrame.pop of tot   -0.306218
         pop    0.298763
         sos    0.150845
         mom    0.729210
         dtype: float64>
```

In[10]中创建了一个新的 Series 对象，只是它的索引有点特殊规律。然后用“.”符号的方式获取某个索引的数据，s.tot 没有问题，但是 s.pop 就有问题了（仔细观察 Out[11]返回的结果），没有得到变量 s 引用的 Series 对象中索引“pop”所对应的值。为什么？用 dir(s)查看，不难发现在结果中也有名称为 pop 的方法，所以 Out[11]得到的其实是一个绑定方法的对象。这也是笔者在前面不推荐使用 In[9]方式的原因，如果遇到了上面所举示例的情况，以“.”方式得到索引的值的目标就无法实现。干脆用 s['pop']吧，万无一失。

再审视 In[3]中的操作，并回忆数组中对下标的解释。如果我们将 Series 对象看成类一维数组的对象（这是有着足够理由的），那么就可以向[ ]中放置不同的下标。

```
In [12]: gdp[['suzhou', 'shanghai', 'beijing']]
Out[12]: suzhou     15475.09
         shanghai    27466.15
         beijing     24899.30
         dtype: float64
```

gdp[['suzhou', 'shanghai', 'beijing']]的下标是由标签索引组成的列表，得到了相应数据组成的新的 Series 对象。

```
In [13]: gdp[gdp>20000]
Out[13]: shanghai    27466.15
         beijing     24899.30
         dtype: float64
```

前面已经说过，Pandas 继承了 NumPy，并且在其基础上对某些功能进行了拓展。所以，读者已经掌握的有关数组的知识都可以拿过来用一用，例如 Series 对象类一维数组。

```
In [14]: g = gdp['tianjin': 'suzhou']
         g
```

```
Out[14]: tianjin     17885.39
         chongqing   17558.76
         suzhou      15475.09
         dtype: float64
```

但是，在某些地方还是存在区别的。例如 In[14]的操作，如果按照对数组的理解，就应该遵守“前包括，后不包括”的一般 Python 序列的切片规则，而这里没有，居然也包括了结束的标签索引的值。

还有一点区别，也是读者要注意的。

```
In [15]: g['wuhan'] = 11912.6
         g
Out[15]: tianjin     17885.39
         chongqing   17558.76
         suzhou      15475.09
         wuhan       11912.60
         dtype: float64

In [16]: gdp
Out[16]: shanghai     27466.15
         beijing      24899.30
         guangzhou    19610.90
         shenzhen     19492.40
         tianjin      17885.39
         chongqing    17558.76
         suzhou       15475.09
         chengdu      12170.20
         hangzhou     11050.50
         dtype: float64
```

观察上面的操作，是否看出什么奥秘了？

In[15]的操作修改了变量 g 引用的 Series 对象，此对象是从 gdp 引用的 Series 对象中切片出来的，但是 gdp 并没有发生变化，这说明在 Pandas 中没有了 NumPy 中所谓的“公用视图”的概念，每次切出来的那部分，之后都生成了一个新对象。

新奇还在继续。

对于 Series 对象，拥有了标签索引的同时，位置索引也没有失效，所以依然可以通过位置索引获取元素，跟 NumPy 中的数组一样。

```
In [16]: gdp[2]
Out[16]: 19610.900000000001   #为什么是这种显示结果？请参考《跟老齐学 Python：轻松入门》
```

由上面示例可知，完全可以如同数组那样在 Series 对象的下标中使用位置索引，甚至切片操作的结果都一样。

```
In [17]: gdp[2: 6]
Out[17]: guangzhou    19610.90
         shenzhen     19492.40
         tianjin      17885.39
         chongqing    17558.76
         dtype: float64
```

In[17]与 In[14]比较，差别就在于对结束值的处理上，In[17]中使用位置索引，又恢复了“前包括，后不包括”的老习惯。

依据以上方式，貌似对 Series 类型的数据进行各种切片操作都已经“没毛病”了。

在软件公司，有一个很重要的工种——测试，一般小妹妹们喜欢从事此工作。可不要小看平时温柔的测试小妹妹，她们常常能够“挖”出程序中的很多 bug，让那些不可一世的程序员汗颜。在测试中，一个被经常使用的方法就是创造一些极端的、特殊的环境，看看程序能不能正常运转。在这里我们就学习测试小妹妹们的做法，构造一个有点特殊的 Series 数据，看看上述切片操作是否还能正常使用。

```
In [18]: s = pd.Series(np.random.randn(4), index=[1, 3, 5, 7])
         s
Out[18]: 1   -1.413869
         3    0.306255
         5   -0.543190
         7   -0.175750
         dtype: float64

In [19]: s[1]
Out[19]: -1.413869134529427

In [20]: s[1: 3]
Out[20]: 3    0.306255
         5   -0.543190
         dtype: float64
```

s 引用的是一个有些奇怪的 Series 对象，它的标签索引居然不是前面见惯了的字符串，而是整数——这与位置索引数据类型相同，从而带来了一些混乱。s[1]中的 1 是什么索引？从结果中推断应该是标签索引；s[1: 3]中的数字呢？还是标签索引吗？从结果推测应该是位置索引。

“贵圈真乱”。

Pandas 是不允许有这种混乱存在的，于是针对索引，有了专门的方法。

- series.iloc[ ]

```
In [21]: s.iloc[1]
Out[21]: 0.306254828661082

In [22]: s.iloc[1: 3]
Out[22]: 3    0.306255
         5   -0.543190
         dtype: float64
```

在 series.iloc[ ]的[ ]里面，只接收位置索引，从上面的演示中也能看出来。对于 Series 对象，若使用位置索引读取某些值，提倡使用 series.iloc[ ]。当然，在不引起误解的情况下，类似 s[1: 3]也是可以使用的。

- series.loc[ ]

```
In [23]: s.loc[1: 3]
Out[23]: 1   -1.413869
         3    0.306255
```

```
        dtype: float64
```

与前面对照，就好理解了，series.loc[ ]的[ ]里面接收的是标签索引，也仅如此。

这样，用上面两种方式就避免了混乱。不过，如果数据对象不会如上面例子那样引起混乱，则前面所说的切片方法依然可以使用。

.loc、.iloc 不仅适用于 Series 对象，还适用于 DataFrame 对象。

通过以上操作，读者应该理解对 Series 类型的数据进行切片的方法了。不过，还没完，因为我们刚刚练习的仅局限于 Index 对象，上一节还专门针对索引对象进行了学习，不要忘记还有 MultiIndex 对象（如果现在就忘记了，请立刻回头复习）。

```
In [24]: gdp_index = [("shanghai", 2015), ("shanghai", 2016), ("beijing", 2015),
                      ("beijing", 2016), ("guangzhou", 2015), ("guangzhou", 2016)]
         gdp_mind = pd.MultiIndex.from_tuples(gdp_index)
         gdp = pd.Series([25300, 27466, 23000, 24899, 18100, 19611], index=gdp_mind)
         gdp
Out[24]: shanghai   2015    25300
                    2016    27466
         beijing    2015    23000
                    2016    24899
         guangzhou  2015    18100
                    2016    19611
         dtype: int64
```

以上是复习，依然使用我们已经熟悉的 gdp 对象。下面演示根据 MultiIndex 对象切片。

```
In [25]: g1 = gdp['shanghai']
         type(g1)
Out[25]: pandas.core.series.Series

In [26]: g1
Out[26]: 2015    25300
         2016    27466
         dtype: int64
```

gdp['shanghai']中的下标是 MultiIndex 对象的第 0 级索引，返回的结果是一个 Series 对象（如 Out[26]所示），其索引为原 MultiIndex 索引中与 0 级相对应的 1 级索引，即符合 MultiIndex 索引对应关系。

刚刚使用过的.loc[]方式对于 MultiIndex 依然适用，如下所示。

```
In [27]: gdp.loc["shanghai"]
Out[27]: 2015    25300
         2016    27466
         dtype: int64
```

.iloc[ ]返回的是第一个数值，而不是 0 级索引中的第一个索引对应的结果，请注意。

```
In [28]: gdp.iloc[0]     #等同于 gdp[0]
Out[28]: 25300
```

接下来就要深入到第 1 级索引了。

```
In [29]: gdp.loc['shanghai', 2015]
```

```
Out[29]: 25300
```

观察 In[29]中的指令格式，在[ ]中，“,”前面是第 0 级的标签索引，后面是第一级的标签索引，通过两级索引最终锁定了它们对应的值。通过 In[29]得到了一个城市某一年的数据，如果要得到各个城市某一年（如 2015 年）的所有数据，是不是可以用 gdp[2015]呢？在做这个操作之前，请读者仔细想一想，可以吗？再回头看一眼 In[25]，是不是两者有冲突？Pandas 会认为此处的“2015”是第 0 级索引。

```
In [30]: gdp[2015]
      …    #省略报错信息中的很多内容，只看最后一行，如下
      IndexError: index out of bounds
```

操作结果告诉我们，“2015”不在索引范围之内——Pandas 认为我们在 0 级索引中找“2015”这个标签索引呢。正确做法是什么？其实 In[29]已经给了提示，就是在[ ]中，“,”前面是针对 0 级的操作，那么现在我们需要 0 级的全部索引，利用从 Python 到 NumPy 的索引和切片的知识及经验，如果要表示全部索引，可以使用一个“:”，前后不用写任何东西（参见《跟老齐学 Python：轻松入门》有关内容），下面就根据此经验试一试。

```
In [31]: gdp.loc[:, 2015]
Out[31]: shanghai     25300
         beijing      23000
         guangzhou    18100
         dtype: int64
```

果然，老马识途，“:”依然好用。

继续测试“经验主义”是否具有普适性。

```
In [32]: gdp.loc[gdp > 19000]
Out[32]: shanghai   2015    25300
                    2016    27466
         beijing    2015    23000
                    2016    24899
         guangzhou  2016    19611
         dtype: int64
```

以上几项都成功了，肯定也会有不成功的，读者不妨自己继续检验。

对于.iloc[ ]，在多级索引中，依然有很大用途。因为在[ ]里面需要填写的是位置索引，而位置索引是不区分级别的，它只是标识了 Series 中每个元素的位置，对于 gdp 所引用的对象而言，就是 gdp.values 的索引。

```
In [33]: gdp.values
Out[33]: array([25300, 27466, 23000, 24899, 18100, 19611])
```

这个索引与在数组中学习的索引是一样的，所以还可以像下面这样取值。

```
In [34]: gdp.iloc[1: 5]
Out[34]: shanghai    2016    27466
         beijing     2015    23000
                     2016    24899
         guangzhou   2015    18100
         dtype: int64
```

```
In [35]: gdp.iloc[[1, 3, 5]]
Out[35]: shanghai    2016    27466
         beijing     2016    24899
         guangzhou   2016    19611
         dtype: int64
```

.iloc[ ]和 loc[ ]分别提供了两个途径，允许在索引上分别使用位置索引和标签索引，为我们从 Series 对象中取值提供了多种途径。

2．DataFrame 对象

DataFrame 是二维的 Pandas 数据，可以在一定程度上类比二维数组，所以建议读者简要复习二维数组的有关索引和切片的知识，以便深刻理解下述内容。

根据前面的知识，我们知道，DataFrame 对象的每列都是 Series 对象，并且这些 Series 对象公用同一个索引。

```
In [36]: population = pd.Series([2415.27, 2151.6, 1270.08], index=["shanghai", "beijing",
                                 "guangzhou"])
         gdp = pd.Series([27466, 24899, 19611], index=["shanghai", "beijing", "guangzhou"])
         d = pd.DataFrame({'gdp':gdp, 'pop':population})
         d
Out[36]:
                       gdp       pop
         shanghai    27466    2415.27
         beijing     24899    2151.60
         guangzhou   19611    1270.08
```

刚才提到，DataFrame 对象是类二维数组，其实撇开索引，其元素就是一个二维数组。

```
In [37]: d.values
Out[37]: array([[ 27466.  ,   2415.27],
                [ 24899.  ,   2151.6 ],
                [ 19611.  ,   1270.08]])
```

二维数组中的某些方法，对 DataFrame 也是适用的，比如下面几个操作。

```
In [38]: d.values[0]
Out[38]: array([ 27466.  ,   2415.27])

In [39]: d.T
Out[39]:
              shanghai  beijing  guangzhou
         gdp  27466.00  24899.0   19611.00
         pop   2415.27   2151.6    1270.08
```

因为读者已经有了二维数组的基础，所以这里不一一列举。

但是，还是需要注意区别的。

```
In [40]: d['gdp']
Out[40]: shanghai     27466
         beijing      24899
         guangzhou    19611
```

```
        Name: gdp, dtype: int64
```

In[40]的操作似乎平淡无奇，实则与以往大不一样。在 Series 对象中，[ ]里面是 Index 对象中的值，而 d['gdp']中的“gdp”是 DataFrame 列（索引）的名称。如果按照数据库表结构的名称来说，就是字段名称。如果修改为 Index 对象中的值会如何？

```
In [41]: d['shanghai']
         …    #报错信息略过
         KeyError: 'shanghai'
```

思考一下，还是能理解的。

Series 对象是一维的，它只有一个方向的索引，所以使用类似 In[40]中的操作，意义非常明确。而 DataFrame 则不然，它有两个维度。于是就规定，类似 In[40]中那样的操作，是专门用来读取列的，即在[ ]中放置的是字段名称（数据库表的术语），或者说是列的索引的名称（d[1]可以吗？建议读者自行测试和思考），没有很明确的指向，有歧义，Pandas 不能推断我们到底想干什么，于是就报错。

前面使用.iloc[ ]和 loc[ ]分别操作位置索引和标签索引，不仅适用于 Series 对象，也适用于 DataFrame 对象。

```
In [42]: d.iloc[1]
Out[42]: gdp    24899.0
         pop     2151.6
         Name: beijing, dtype: float64

In [43]: d.iloc[1, 1]
Out[43]: 2151.5999999999999

In [44]: d.iloc[1:3, :2]
Out[44]:
                      gdp      pop
         beijing     24899   2151.60
         guangzhou  19611   1270.08
```

.iloc[ ]接收的是位置索引，如上面的示例。这时候，你面前的 DataFrame 对象就完全变成了一个二维数组，所有关于二维数组中的索引和切片的方法和规则，都可以通过.iloc[ ]对 DataFrame 对象使用——所以要复习。

```
In [45]: d.loc["beijing", "pop"]
Out[45]: 2151.5999999999999

In [46]: d.loc["beijing":"guangzhou", "pop"]
Out[46]: beijing     2151.60
         guangzhou    1270.08
         Name: pop, dtype: float64

In [47]: d.loc["beijing":"guangzhou", "gdp":"pop"]
Out[47]:
                      gdp     pop
         beijing     24899  2151.60
         guangzhou  19611  1270.08
```

.loc[ ]接收的是标签索引，注意使用标签索引（不论是 0 轴方向还是 1 轴方向，借用数组的描述方法）切片的原则是“前包括，后也包括”——还是老样子。

除上面两种方法外，还有一个名为.ix[ ]的获得索引切片的方法，它是上述.loc[ ]和.iloc[ ]的混合方案，即在[ ]中可以混合放置位置索引和标签索引，如下所示。

```
In [48]: d.ix[:2, "pop"]
         /usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:1:
         DeprecationWarning:
         .ix is deprecated. Please use
         .loc for label based indexing or
         .iloc for positional indexing

         See the documentation here:
         http://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate_ix
         """Entry point for launching an IPython kernel.
Out[48]: shanghai    2415.27
         beijing     2151.60
         Name: pop, dtype: float64
```

注意阅读提示信息。虽然没有报错，但是告诉我们“.ix is deprecated”。因为在一些使用旧版本的资料中，读者会看到 ix[ ]，所以这里也提示一下。

综上所述，对于 DataFrame 对象而言，我们可以简单地总结一下。

- 如果仅取某列数据，则可以根据列名称或者字段名称直接获得。

```
In [49]: d['pop']
Out[49]: shanghai     2415.27
         beijing      2151.60
         guangzhou    1270.08
         Name: pop, dtype: float64
```

- 如果要获得某些行的切片，则可以用下述多种方式，在具体操作中可以根据实际情况选用。

```
In [50]: d["beijing" : "guangzhou"]
Out[50]:
                     gdp      pop
         beijing    24899  2151.60
         guangzhou  19611  1270.08

In [51]: d.loc["beijing":"guangzhou"]
Out[51]:
                     gdp      pop
         beijing    24899  2151.60
         guangzhou  19611  1270.08

In [52]: d.iloc[1:]
Out[52]:
                     gdp      pop
         beijing    24899  2151.60
         guangzhou  19611  1270.08
```

- 如果要通过行和列的共同约束（包括索引和切片）获得某些数据，则要使用 DataFrame

对象的.loc[ ]、.iloc[ ]属性进行操作。事实上，这些属性操作不仅局限于此，它们的使用范围比较广。

```
In [53]: d.loc["shanghai", "pop"]
Out[53]: 2415.27

In [54]: d.loc[:, 'pop']
Out[54]: shanghai    2415.27
         beijing     2151.60
         guangzhou   1270.08
         Name: pop, dtype: float64
In [55]: d.ix[['shanghai', 'guangzhou'], "pop"]
Out[55]: shanghai    2415.27
         guangzhou   1270.08
         Name: pop, dtype: float64
```

以上我们学习了 Index 对象作为 DataFrame 数据索引的部分数据的读取方法，下面要学习索引对象为 MultiIndex 类型的数据读取方法。

为了更好地演示，先建立一个典型的 DataFrame 对象，它的行和列都以 MultiIndex 对象为索引。

```
In [56]: mind = pd.MultiIndex.from_product([[2016, 2017], [1, 2]], names=["year",
                                            "test"])
         columns = pd.MultiIndex.from_product([['Hertz', 'Newton', 'Sola'], ['Chinese',
                                              'Phy']],names=['name', 'subject'])
         data = np.round(np.random.randn(4, 6), 1)
         data = data * 10 + 70
         scores = pd.DataFrame(data, index=mind, columns=columns)
         scores
Out[56]:
```

	name	Hertz		Newton		Sola	
	subject	Chinese	Phy	Chinese	Phy	Chinese	Phy
year	test						
2016	1	81.0	80.0	75.0	56.0	69.0	63.0
	2	52.0	79.0	73.0	64.0	98.0	79.0
2017	1	68.0	60.0	72.0	78.0	68.0	73.0
	2	66.0	58.0	52.0	88.0	62.0	72.0

首先看一下如何获取不同列的数据。

```
In [57]: scores['Hertz']
Out[57]:
```

	subject	Chinese	Phy
year	test		
2016	1	81.0	80.0
	2	52.0	79.0
2017	1	68.0	60.0
	2	66.0	58.0

```
In [58]: scores['Hertz', 'Phy']
Out[58]: year   test
         2016    1      80.0
                 2      79.0
```

```
        2017    1      60.0
                2      58.0
      Name: (Hertz, Phy), dtype: float64
```

此外，前面用过的.loc[ ]和.iloc[ ] 在这里依然有用。下面列出几个操作示例，对于更多的操作，读者可以自己尝试。

```
In [59]: scores.loc[:, ('Sola', 'Phy')]
Out[59]: year   test
         2016    1     63.0
                 2     79.0
         2017    1     73.0
                 2     72.0
         Name: (Sola, Phy), dtype: float64

In [60]: scores.iloc[1:3]
Out[60]:
             name      Hertz          Newton          Sola
             subject   Chinese  Phy   Chinese   Phy    Chinese   Phy
         year   test
         2016    2     52.0    79.0    73.0    64.0    98.0    79.0
         2017    1     68.0    60.0    72.0    78.0    68.0    73.0

In [61]: scores.loc[2016, 2]
Out[61]: name    subject
         Hertz   Chinese      52.0
                 Phy          79.0
         Newton  Chinese      73.0
                 Phy          64.0
         Sola    Chinese      98.0
                 Phy          79.0
         Name: (2016, 2), dtype: float64

In [62]: scores.loc[(2016, 2), ('Newton')]
Out[62]: subject
         Chinese  73.0
         Phy      64.0
         Name: (2016, 2), dtype: float64
```

在进行各种索引和切片的操作中，也不都是如上面那样顺利的。比如下面的操作，试图得到所有 test 为 1 的 Phy 的值，首先根据经验来操作。

```
In [63]: scores.loc[(:, 1), (:, "Phy")]
         File "<ipython-input-245-6dc7bcdae446>", line 1
         scores.loc[(:,1), (:,"Phy")]
                      ^
         SyntaxError: invalid syntax
```

报错了。“经验主义”失效了。

Pandas 为此提供了新的解决方案。

```
In [64]: idx = pd.IndexSlice
         scores.loc[idx[:, 1], idx[:, "Phy"]]
```

```
Out[64]:
              name    Hertz Newton  Sola
             subject   Phy    Phy   Phy
        year  test
        2016    1     80.0   56.0  63.0
        2017    1     60.0   78.0  73.0
```

Pandas 为 Series 对象和 DataFrame 对象提供了多种获取部分数据的方法，需要读者多练习，熟悉每种方法的特点。在项目中，可以根据具体项目的条件选择使用。

在本节就要结束的时候，给读者演示一个笔者在调试上面的代码时所遇到的“坑”，注意别掉进去。

```
In [65]: columns = pd.MultiIndex.from_product([['Hertz', 'Newton', 'Sola'], ['Physics',
                                                'Chinese']],
         names=['name', 'subject'])
```

这是笔者一开始创建的 columns，后来掉“坑”里面了，才改用了 In[56]所创建的 columns。不过笔者还是还原一下是怎么掉“坑”里的，请保持一个幸灾乐祸的心态来观看。

```
In [66]: scores = pd.DataFrame(data, index=mind, columns=columns)
```

然后是 In[67]的操作了，“坑”就在这里，请注意。

```
In [67]: scores.loc[idx[:,1], idx[:, "Phy"]]
---------------------------------------------------------------------------
UnsortedIndexError                        Traceback (most recent call last)
…    #省略部分报错信息
UnsortedIndexError: 'MultiIndex Slicing requires the index to be fully lexsorted tuple
len (2), lexsort depth (1)'
```

笔者最终还是“爬”上来了，认真阅读报错信息，这就是笔者能“爬”上来的原因和方法。

“UnsortedIndexError”，这个错误提示已经很明确地告诉我们，错误的根源在于使用了没有排序的索引。检讨自己的操作，发现 In[65]中定义的 columns 的第 1 级别索引没有排序。请谨记，在实施索引和切片的时候，作为索引的 MultiIndex 对象一定要从小到大进行排序。笔者修正了 In[65]的操作，按照 In[56]中的操作实施，最终成功了。

重点不是看笔者掉进“坑”里，而是要学怎么“爬”出来。

2.4　文件读写操作

将“大数据”都放到内存里面，这是很危险的，并且也不现实。一般情况下，各种数据都会保存在文件中，要处理它们的时候，从文件中读出来，处理完之后，还要将结果存入文件。

这里所说的“文件”，当然也包括数据库文件，但是本节不讲解对数据库的读写，因为这方面的知识已经在《跟老齐学 Python：轻松入门》一书中有详细说明了，读者可参考。

1．CSV 文件

为了使读者理解 CSV 文件，读者从《维基百科》的相关词条中摘抄一段文字。

逗号分隔值（Comma-Separated Values，CSV，有时也称为字符分隔值，因为分隔字符也可

以不是逗号），其文件以纯文本形式存储表格数据（数字和文本）。纯文本意味着该文件是一个字符序列，不含必须像二进制数字那样被解读的数据。CSV 文件由任意数目的记录组成，记录间以某种换行符分隔；每条记录由字段组成，字段间的分隔符是其他字符或字符串，最常见的是逗号或制表符。通常，所有记录都有完全相同的字段序列。

由此可知，CSV 文件常常用于保存数据，Pandas 提供了对该文件进行读写操作的方法。

```
In [1]: import numpy as np
        import pandas as pd
```

输入“pd.read_”，然后按 Tab 键，会看到如图 2-4-1 所示的诸多函数，这些函数都是用来从某个类型的文件中读取数据的。

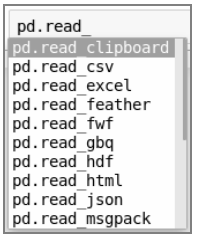


图 2-4-1　读取数据的函数

“弱水三千只取一瓢”，不贪多，就研究 pd.read_csv()的使用吧。其他的函数，如果读者在项目实践中遇到，可以参考下面的学习过程。

继续使用我们已经反复提及的方法，输入“pd.read_csv?”来查看文档，呈现在眼前的文档或许让你有点头晕。

```
pd.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None,
index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True,
dtype=None, engine=None, converters=None, true_values=None, false_values=None,
skipinitialspace=False, skiprows=None, nrows=None, na_values=None,
keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True,
parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None,
dayfirst=False, iterator=False, chunksize=None, compression='infer', thousands=None,
decimal=b'.', lineterminator=None, quotechar='"', quoting=0, escapechar=None,
comment=None, encoding=None, dialect=None, tupleize_cols=False, error_bad_lines=True,
warn_bad_lines=True, skipfooter=0, skip_footer=0, doublequote=True,
delim_whitespace=False, as_recarray=False, compact_ints=False, use_unsigned=False,
low_memory=True, buffer_lines=None, memory_map=False, float_precision=None)
```

这么多参数？！

稍安勿躁。因为不是所有的参数都经常用到，所以表 2-4-1 列出的是笔者认为常用的参数。读者先有一个大概的了解，然后看示例。

表 2-4-1　read_csv()的部分参数解释

参　数	数据类型	说　明
filepath_or_buffer	字符串（文件）或其他类文件对象	文件、文件的 URL、字符串等能够读取的对象地址
sep	字符串	分隔符，默认为“,”，也可以是其他字符串或正则表达式

续表

参　数	数据类型	说　明
delimiter	字符串	同上。在设置分隔符的时候，sep 和 delimiter 二选一使用
header	整数、整数元素的列表	默认为 infer。以整数表示该行作为列标签（字段名称），比如 header=0，意味着数据表的第一行作为列标签；也可以是由整数组成的列表。如果不指定，则表示 None
names	列表和类数组对象	默认为 None。当 header=None 时，以列表指定列的标签
index_col	整数、字符串	默认为 None。以整数或字符串指定某一列或多列作为索引
converters	字典	默认为 None。由列序号或列标签作为 key、函数作为 value 组成的字典，相应列的数据被传入该函数
skiprows	整数、列表	默认为 None。忽略的行号（列表），或者忽略的行数（整数），都是从文件的第一行开始算起，并记为 0
skipfooter	整数	默认为 0。从文件末尾开始算起的忽略的行数
nrows	整数	默认为 None。指定需要读取的行数（从相对文件开始算起）
na_values	数字、字符串、类列表、字典	默认为 None。用于替换 NA 的值
parse_dates	布尔型、列表、字典	默认为 False。如果为 True，则解析索引；如果为列序号或列标签组成的列表，则解析相应的列为日期列。如果是由列表元素组成的列表，比如[[1, 3]]，则将序号为 1、3 的列组合起来解析为一个日期列。如果是类似{'foo' : [1, 3]}样式的字典，则将序号为 1、3 的列解析为一个日期列并命名为'foo'
keep_date_col	布尔型	默认为 False。若 parse_dates 指定了合并的多个列，则 keep_date_col 为 True 时，会保持原有的列
date_parser	函数	默认为 False。指定用于转换日期的函数。在默认情况下使用 dateutil.parser.parser
dayfirst	布尔型	如果为 True，则将日期解析为国际格式，即 DD/MM 格式
iterator	布尔型	默认为 False。如果为 True，则返回 TextFileReader 对象，以便用 get_chunk()函数逐块读取数据
chunksize	整数	默认为 None。指定文件块的大小，返回用于迭代的 TextFileReader 对象
comment	字符串	默认为 None。该字符后面为注释，不作为数据读入。例如 comment="#"，表示#符号之后的内容作为注释被忽略
encoding	字符串	默认为 None。指定编码格式，比如常用的'utf-8'

如果读者需要详细阅读每个参数完整的、原汁原味的说明，可以查看官方文档。下面通过示例演示基本的使用方法（示例中的文件可以在本书的代码仓库中下载）。

```
In [2]: !cat /home/qiwsir/Documents/DataAnalysis/chapter02/gdp-population.csv
        City_Name,GDP,Population
        SHANGHAI,27466.15 ,2419.70
        BEIJING,24899.30 ,2172.90
        GUANGZHOU,19610.90 ,1350.11
        SHENZHEN,19492.60 ,1137.87
        TIANJIN,17885.39 ,1562.12
        CHONGQING,17558.76 ,3016.55
        SUZHOU,15475.09 ,1375.00
```

```
CHENGDU,12170.20 ,1591.76
```

笔者创建了一个 CSV 文件，其中记录了 2016 年我国排名前八的城市 GDP 及其人口数据。在这个文件中，不同数据之间是用逗号（“,”）分隔的，下面使用 pd.read_csv()读取此文件，如下所示。

```
In [3]: gdp = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter02/gdp-
                          population.csv")
        gdp
Out[3]:
             City_Name       GDP      Population
        0    SHANGHAI     27466.15     2419.70
        1    BEIJING      24899.30     2172.90
        2    GUANGZHOU    19610.90     1350.11
        3    SHENZHEN     19492.60     1137.87
        4    TIANJIN      17885.39     1562.12
        5    CHONGQING    17558.76     3016.55
        6    SUZHOU       15475.09     1375.00
        7    CHENGDU      12170.20     1591.76
```

在 In[3]的操作中，没有指定 read_csv()的参数 header 的值，Pandas 根据文件中的数据特点进行了推断，并且将第一行作为列标签。

```
In [4]: gdp.columns
Out[4]: Index(['City_Name', 'GDP', 'Population'], dtype='object')
```

当然，如果使用下面的方式，声明 header=0，效果是相同的。

```
In [5]: gdp = pd.read_csv("/home/qiwsir/Documents/data_analysis/gdp-population.csv",
                          header=0)
```

可能我们需要重命名所有列标签的名称，可以按下面这样做。

```
In [6]: pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter02/gdp-population.csv",
                    names=['CITY', 'GDP', 'POP'])
Out[6]:
              CITY           GDP         POP
        0    City_Name       GDP    Population
        1    SHANGHAI     27466.15     2419.70
        2    BEIJING      24899.30     2172.90
        3    GUANGZHOU    19610.90     1350.11
        4    SHENZHEN     19492.60     1137.87
        5    TIANJIN      17885.39     1562.12
        6    CHONGQING    17558.76     3016.55
        7    SUZHOU       15475.09     1375.00
        8    CHENGDU      12170.20     1591.76
```

具有一双慧眼的读者可能根本不需要看 In[6]的操作，一眼就能看出 Out[6]的结果有问题。原来 CSV 文件中已有的字段名称那一行，在这里被当作一行数据记录了，而事实上是不需要它的。因为列标签的名称已经在参数 names 中重命名了，并且这一行又不是任何城市的 GDP 和人口数量，所以，In[6]错了。

怎么改？

```
In [7]: pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter02/gdp-
```

```
                    population.csv", names=['CITY', 'GDP', 'POP'], skiprows=[0])
Out[7]:
             CITY          GDP         POP
        0    SHANGHAI     27466.15     2419.70
        1    BEIJING      24899.30     2172.90
        2    GUANGZHOU    19610.90     1350.11
        3    SHENZHEN     19492.60     1137.87
        4    TIANJIN      17885.39     1562.12
        5    CHONGQING    17558.76     3016.55
        6    SUZHOU       15475.09     1375.00
        7    CHENGDU      12170.20     1591.76
```

使用参数 skiprows，将那一行忽略。

再观察上面的数据结果，如果把城市名称作为返回的 DataFrame 对象的索引，就更好了。

```
In [8]: gdp = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter02/gdp-
                          population.csv", names=['CITY', 'GDP', 'POP'], skiprows=[0],
                          index_col=0)
        gdp.index
Out[8]: Index(['SHANGHAI', 'BEIJING', 'GUANGZHOU', 'SHENZHEN', 'TIANJIN', 'CHONGQING',
               'SUZHOU', 'CHENGDU'],
              dtype='object', name='CITY')
```

比较完美地解决了提出的要求。

以上是一个比较理想化的数据文件，在实际的工作中，可能会因为某种原因，有些数据缺失了。在数据分析中，对于缺失的数据要做专门处理，后面会单独介绍如何处理缺失的数据。这里先看一下在读取文件的时候如何处理缺失的数据。

```
In [9]: !cat /home/qiwsir/Documents/DataAnalysis/chapter02/gdp-pop.csv
        City_Name,GDP,Population
        SHANGHAI,27466.15,
        BEIJING,24899.3,2172.9
        GUANGZHOU,19610.9,1350.11
        SHENZHEN,,1137.87
        TIANJIN,17885.39,1562.12
        CHONGQING,17558.76,3016.55
        SUZHOU,15475.09,1375
        CHENGDU,12170.2,1591.76
```

上面显示的数据并不是非常完整，有缺失，能用你的慧眼看出来吗？如果这个文件使用 read_csv()函数来读取，Pandas 会如何处理缺失的数据？不用猜，上代码，看结果。

```
In [10]: pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter02/gdp-pop.csv ",
                     names=['CITY', 'GDP', 'POP'], skiprows=[0], index_col=0)
Out[10]:
                        GDP         POP
         CITY
         SHANGHAI     27466.15     NaN
         BEIJING      24899.30     2172.90
         GUANGZHOU    19610.90     1350.11
         SHENZHEN     NaN          1137.87
         TIANJIN      17885.39     1562.12
```

```
        CHONGQING    17558.76     3016.55
        SUZHOU       15475.09     1375.00
        CHENGDU      12170.20     1591.76
```

Pandas 对缺失的数据做了自动处理，以 NaN 表示。当然，情况可能还会恶化，比如出现了下面的数据文件，不但数据缺失，而且在源数据中缺失的元素用了不同的符号表示。

```
In [11]: !cat /home/qiwsir/Documents/DataAnalysis/chapter02/gdp-pop2.csv
         City_Name,GDP,Population
         SHANGHAI,27466.15,
         BEIJING,24899.3,2172.9
         GUANGZHOU,19610.9,1350.11
         SHENZHEN,None,1137.87
         TIANJIN,17885.39,1562.12
         CHONGQING,17558.76,NA
         SUZHOU,15475.09,1375
         CHENGDU,12170.2,1591.76
```

又在考察你是否具有像雄鹰那样敏锐的观察力了。文件中除有一处缺失数据外，另外两处分别用 None 和 NA 表示了缺失部分——NA 是表示缺失数据的一种方式。

```
In [12]: s = {"GDP":[None]}
         pd.read_csv("/home/qiwsir/Documents/data_analysis/gdp-pop.csv",
                     names=['CITY', 'GDP', 'POP'], skiprows=[0], index_col=0, na_values=s)
Out[12]:
        CITY          GDP          POP
        SHANGHAI      27466.15     NaN
        BEIJING       24899.30     2172.90
        GUANGZHOU     19610.90     1350.11
        SHENZHEN      NaN          1137.87
        TIANJIN       17885.39     1562.12
        CHONGQING     17558.76     NaN
        SUZHOU        15475.09     1375.00
        CHENGDU       12170.20     1591.76
```

Pandas 也提供了很多方法解决此问题。请注意 In[12]的操作，在 Python 中 None 是一个对象，而 Pandas 中的 NaN 是缺失数据的标记，两者的区别要清楚。

以上操作是将文件内容一次性读取。当文件中的数据量很大或者不需要全部读取的时候，可以通过 nrows 参数设置读入几行，或者使用 chunksize 分块读入。

```
In [15]: pd.read_csv("/home/qiwsir/Documents/data_analysis/gdp-pop.csv", nrows=3)
Out[15]:
            City_Name     GDP          Population
        0   SHANGHAI      27466.15     NaN
        1   BEIJING       24899.30     2172.90
        2   GUANGZHOU     19610.90     1350.11
```

为了演示 chunksize 参数的作用，笔者不辞劳苦地从生态环境部的网站（http://www.mep.gov.cn/）上复制了 2017 年 9 月 19 日部分城市的 AQI 指数，并保存为 CSV 文件。

```
In [16]: aqi_chunk = pd.read_csv("/home/qiwsir/Documents/data_analysis/aqi.csv",
    .                            chunksize=50,
    .                            encoding="utf-8", header=0)
```

```
        aqi_chunk
Out[16]: <pandas.io.parsers.TextFileReader at 0x7fea796095c0>
```

通过 In[16]的操作，最终得到了一个 TextFileReader 对象。这个对象是可迭代的，其迭代单元的数据记录行数由 chunksize=50 来确定（在真实的项目中，这个数字一般不会这么小，比如是 10000）。之后，就可以用 for 循环来完成对迭代对象的操作了。

```
In [17]: tot = pd.Series([])
         for city in aqi_chunk:
             tot = tot.add(city['级别'].value_counts(), fill_value=0)
         tot
Out[17]: 中度污染     8.0
         优         105.0
         良         180.0
         轻度污染    62.0
         dtype: float64
```

创建一个空的 Series 对象，然后对所有城市的空气质量按照“级别”字段（列标签）进行统计（该文件的“级别”字段中共有 4 类数据，即优、良、轻度污染、中度污染），得到了不同等级的城市数量，最后添加到 Series 对象中，即 In[17]的操作。

除从 CSV 文件中读数据外，还可以将数据写入 CSV 文件中，这个操作使用的是 pd.DataFrame.to_csv()函数，此函数的具体使用方法可以参考 2.1 节中的有关叙述，此处不再重复。

2．HDF5 文件

HDF5，这是什么文件？或许对一些读者来讲，这是一个新东西。一般来讲，笔者遇到新东西，喜欢到《维基百科》上看看，或者在 Google 里搜索一下，读者不妨也这么试试。下面的说明就来自于《维基百科》。

HDF（Hierarchical Data Format）指一种为存储和处理大容量科学数据设计的文件格式及相应库文件。HDF 最早由 NCSA 开发，目前在非盈利组织 HDF 小组的维护下继续发展。当前流行的版本是 HDF5。

HDF5 文件包含两种基本数据对象。

- 群组（group）：类似文件夹，可以包含多个数据集或下级群组。
- 数据集（dataset）：数据内容，可以是多维数组，也可以是更复杂的数据类型。

HDF5 文件在速度、内存占有、压缩程度等方面比一般常用的文件类型具有优势，所以，在存储大数据的时候，HDF5 文件常常是不二之选。

Python 对 HDF5 文件有很好的支持——h5py，推荐读者查看官方文档（http://docs.h5py.org/en/latest/）。下面先安装 h5py，然后通过 Python 对其进行讲解。安装 h5py 的命令如下。

```
$ sudo pip3 install h5py
```

这种安装第三方库的方法是我们常用的，读者应该熟悉了。安装成功之后，进入到 Python 交互模式。

```
>>> import h5py
```

正确引入，若不报错，则说明安装好了。

下面创建一个扩展名为.hdf5 的文件，用来保存数据。

```
>>> f = h5py.File("hdftest.hdf5", "w")
>>> f
<HDF5 file "hdftest.hdf5" (mode r+)>
```

文件对象创建之后，就可以对这个对象进行操作了。依据在 Python 基础知识学习中获得的基本思维方法（如果不理解，请务必阅读《跟老齐学 Python：轻松入门》），应该看一下这个对象的属性和方法，才能知道它是什么，以及可以干什么。

```
>>> dir(f)
['_MutableMapping__marker', '__abstractmethods__', '__bool__', '__class__',
'__contains__', '__delattr__', '__delitem__', '__dict__', '__dir__', '__doc__',
'__enter__', '__eq__', '__exit__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__module__', '__ne__', '__new__', '__nonzero__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__slots__',
'__str__', '__subclasshook__', '__weakref__', '_abc_cache', '_abc_negative_cache',
'_abc_negative_cache_version', '_abc_registry', '_d', '_e', '_id', '_lapl', '_lcpl',
'attrs', 'clear', 'close', 'copy', 'create_dataset', 'create_group', 'driver', 'fid',
'file', 'filename', 'flush', 'get', 'id', 'items', 'keys', 'libver', 'mode', 'move',
'name', 'parent', 'pop', 'popitem', 'ref', 'regionref', 'require_dataset',
'require_group', 'setdefault', 'update', 'userblock_size', 'values', 'visit',
'visititems']
```

在引用的《维基百科》关于 HDF5 文件介绍的文字中，我们已经了解到，这种类型的文件中包含 group 和 dataset 两种基本数据对象。再看上面的显示结果，有两个方法是我们必须要研究的，即 create_group 和 create_dataset，重要的学习方法是使用 help()，请读者自行操作。

接着在 f 文件中创建 dataset，基本操作方法如下：

```
>>> d = f.create_dataset(name='/dataset1', shape=(99, 99))
>>> d
<HDF5 dataset "dataset1": shape (99, 99), type "<f4">
>>> d.name
'/dataset1'
>>> d.shape
(99, 99)
>>> d.dtype
dtype('float32')
>>> d[:]
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]], dtype=float32)
```

然后就是创建 group 了，当然要使用 create_group()，简要操作如下：

```
>>> g = f.create_group("group1")
```

```
>>> g.name
'/group1'
```

接着在上面所创建的 group 中增加 dataset。

```
>>> d2 = g.create_dataset(name="in_group1", shape=(20,))
>>> d2.name
'/group1/in_group1'
```

HDF5 文件也是一个类字典对象，拥有类似字典的所有操作。

```
>>> d3 = f['group1/in_group1']
>>> d3
<HDF5 dataset "in_group1": shape (20,), type "<f4">
```

跟字典中通过 key 读取 value 一样，d3 所引用的对象就是前面所创建的 dataset。

```
>>> list(f.keys())
['dataset1', 'group1']
>>> list(f.items())
[('dataset1', <HDF5 dataset "dataset1": shape (99, 99), type "<f4">), ('group1', <HDF5
group "/group1" (1 members)>)]
>>> list(f.values())
[<HDF5 dataset "dataset1": shape (99, 99), type "<f4">, <HDF5 group "/group1" (1 members)>]
```

真的可以把 f 看作字典。

在 Python 中，除 h5py 外，还有 PyTables（官方网站：http://www.pytables.org/），这里不再介绍 PyTables 了，读者完全有能力根据官方网站进行学习。

以上仅仅是简要介绍在 Python 中如何操作 HDF5 文件。这不是本节的重点，重点还在下面，就是使用 Pandas 对此类文件进行读写。

在 Pandas 中有一个名为 HDFStore 的类，这个类通过 PyTables 存储 Pandas 对象。所以，还要先安装 PyTables。安装 PyTables 的命令如下。

```
$ sudo pip3 install tables
```

安装之后，回到 Jupyter 中，通过实例化 HDFStore()类创建 HDF5 文件对象。

```
In [18]: store = pd.HDFStore('/home/qiwsir/Documents/data_analysis/data.h5')
```

为了向 store 对象中添加数据，再次调用原有的 CSV 文件读取数据。

```
In [19]: gdp = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter02/gdp-
                           population.csv", names=['CITY', 'GDP', 'POP'], skiprows=[0],
                           index_col=0)
```

接下来的操作就回到了类字典方式了。

```
In [20]: store['gdp'] = gdp['GDP']
         store['gdp_pop'] = gdp
         store
Out[20]: <class 'pandas.io.pytables.HDFStore'>
         File path: /home/qiwsir/Documents/data_analysis/data.h5
         /gdp              series      (shape->[8])
         /gdp_pop          frame       (shape->[8,2])
```

```
In [21]: store['gdp_pop']
Out[21]:
         CITY          GDP       POP
         SHANGHAI      27466.15  2419.70
         BEIJING       24899.30  2172.90
         GUANGZHOU     19610.90  1350.11
         SHENZHEN      19492.60  1137.87
         TIANJIN       17885.39  1562.12
         CHONGQING     17558.76  3016.55
         SUZHOU        15475.09  1375.00
         CHENGDU       12170.20  1591.76
```

当读者观察了上述操作之后，笔者还要再次强调，对于 HDF5 文件对象，可以把它看作字典，所有针对字典的操作都能拿过来使用。

最后提示，HDF5 不是数据库，因此不要把它当作数据库来使用，特别是对 HDF5 文件进行多线程或多进程写入的时候，难免会出错，所以尽可能将 HDF5 文件应用在写入较少、读取较多的情况中——“一次写、多次读”。

对于数据存储的文件，除 CSV 和 HDF5 文件外，还有很多别的文件类型，比如电子表格文件、保存为 JSON 或 XML 的数据文件、数据库文件等。对这些文件的读写，一方面可以使用 Python 中的方法实现，另一方面 Pandas 也提供了必要的函数，例如下面的函数。

- pd.ExcelFile()：读取 Microsoft Excel 文件。
- pd.io.sql：此模块下有用于数据库文件读写方法，如 read_sql()和 to_sql()。

Pandas 能够对多种文件进行有效的读写操作。它的功能不仅如此，还有很多，旨在让数据处理的操作更便捷。

2.5 处理缺失数据

缺失数据在数据分析中是司空见惯的，对其不可熟视无睹，要进行适当处理。从语法的角度看，对“缺失数据”这个短语有两种不同的理解，一是认为它是一个动宾短语，即表达了“数据表中缺少了数据”；二是认为它是“缺失的数据”，其实是一个名词前面加了形容词。汉语有歧义。英语怎么说？Missing Data。所以，此处的“缺失数据”实为一个数据。

1．NumPy 中的缺失数据

在 Python 中，我们遇到过表示“无、空、没有”的对象。例如，有的函数没有返回值，但事实上是 return None，用 None 表示“空”这个对象。正如笔者在《跟老齐学 Python：轻松入门》一书中强调过的，“空”不是什么都没有，“空”也是一个对象。Python 中万物皆对象，“空”必然是对象。解释这么多，概括为一句：“空”很有名，None 是对象。

在数据中，我们也会常常遇到类似的问题，比如由于某种原因，数据中出现了“空缺”，那么这个位置就可以用 None 表示了。

```
In [1]: import numpy as np
        import pandas as pd
In [2]: a = np.array([1, None, 5, 7, 9])
        b = np.array([1, 3, 5, 7, 9])
```

```
        a.dtype
Out[2]: dtype('O')

In [3]: b.dtype
Out[3]: dtype('int64')
```

因为 a 中有一个 None，所以对该数组元素类型进行判断的时候，返回的是 object。如果对数组 a 进行运算，肯定是要报错的。

```
In [4]: a.sum()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-fe98c1bd5fd4> in <module>()
----> 1 a.sum()

/usr/local/lib/python3.5/dist-packages/numpy/core/_methods.py in _sum(a, axis, dtype,
out, keepdims)
     30
     31 def _sum(a, axis=None, dtype=None, out=None, keepdims=False):
---> 32     return umr_sum(a, axis, dtype, out, keepdims)
     33
     34 def _prod(a, axis=None, dtype=None, out=None, keepdims=False):

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

在现实中，数据中有“空缺”是常见的现象。难道就因为这个原因，该数据就不能直接被操作了吗？NumPy 提供了另外一个表示空缺的数据 np.nan。

```
In [5]: c = np.array([1, np.nan, 5, 7, 9])
        c.dtype
Out[5]: dtype('float64')
```

对比 c.dtype 和 a.dtype 的结果，None 和 np.nan 是两种完全不同类型的数据。

```
In [6]: np.nan?
Type:       float
String form: nan
Docstring:
float(x) -> floating point number

Convert a string or number to a floating point number, if possible.
```

原来 np.nan 是一个浮点数，只不过这个浮点数是用来标记缺失数据的，由此也就理解 c.dtype 的返回结果了。也正如此，np.nan 才能够像一个数字那样参与各种运算，例如下面的示例。

```
In [7]: 7 + np.nan
Out[7]: nan

In [8]: 0 * np.nan
Out[8]: nan

In [9]: c.sum()
Out[9]: nan
```

在小学数学中，学习过“0 乘以任何数都为 0”，当 0 遇到“空”时，这个规则被颠覆了。不过也没有什么太稀奇的，np.nan 虽然是浮点数，但表示的是缺失数据。正是有了这种表示缺失数据的方式，才能够让运算得以进行。

对于 In[5]中所创建的数组，如果以 In[9]来计算，得到的结果是 nan，还是缺失数据，这似乎也没有真正解决问题。难道说一个班级里面，会因为一个学生缺考而全班总分也“缺失”了吗？肯定不是的，NumPy 中有专门函数处理这种情况。

```
In [10]: np.nansum(c)
Out[10]: 22.0

In [11]: np.nanmean(c)
Out[11]: 5.5
```

借助于 NumPy 中的 np.nan，我们引入了一个新的浮点数，一般的表示可以用 NaN，其含义是 Not a Number，翻译为汉语是“不是一个数”，但却是一类表示未定义或不可表示的数值。

2．Pandas 处理缺失数据

在 NumPy 中对 None 的处理总感觉不是很智能，于是 Pandas 来解决这个问题了。

```
In [12]: s = pd.Series([1, np.nan, 3, None, 7, 9])
         s
Out[12]: 0   1.0
         1   NaN
         2   3.0
         3   NaN
         4   7.0
         5   9.0
         dtype: float64
```

尽管在创建 Series 对象的列表中有 None，但在最终的 Series 中却被转换为 NaN，这样就不妨碍计算了，并且在计算时，Pandas 自动摒除了 NaN 数据。

```
In [13]: s.sum()
Out[13]: 20.0

In [14]: s[0] = None    #Pandas 对 None 对象进行转换
         s
Out[14]: 0   NaN
         1   NaN
         2   3.0
         3   NaN
         4   7.0
         5   9.0
         dtype: float64
```

Pandas 还提供了 4 个针对缺失数据进行操作的函数，它们分别是 isnull()、notnull()、dropna()和 fillna()。

```
In [15]: s.isnull()
Out[15]: 0    True
         1    True
         2   False
```

```
         3    True
         4   False
         5   False
         dtype: bool

In [16]: s.notnull()
Out[16]: 0   False
         1   False
         2    True
         3   False
         4    True
         5    True
         dtype: bool

In [17]: s[s.notnull()]
Out[17]: 2   3.0
         4   7.0
         5   9.0
         dtype: float64
```

isnull()和 notnull()返回的是布尔型数据，用返回的对象作为下标，可以对原有对象进行筛选，如 In[17]所示。

```
In [18]: s.dropna()
Out[18]: 2   3.0
         4   7.0
         5   9.0
         dtype: float64

In [19]: s
Out[19]: 0   NaN
         1   NaN
         2   3.0
         3   NaN
         4   7.0
         5   9.0
         dtype: float64
```

dropna()的作用是删除所有 NaN 数据，并返回不含 NaN 的新对象，注意原有的对象没有受到影响。对于 Series 对象比较好理解，如果是 DataFrame 对象，应该怎样“删除”呢？是删除行，还是删除列？

```
In [20]: df = pd.DataFrame([[1, 2, np.nan, 4], [5, None, 7, 8], [9, 10, 11, 12]])
         df
Out[20]:
            0     1     2   3
         0  1   2.0   NaN   4
         1  5   NaN   7.0   8
         2  9  10.0  11.0  12

In [21]: df.dropna()
Out[21]:
            0     1     2   3
```

```
         2  9  10.0  11.0  12

In [22]: df.dropna(axis=1)
Out[22]:
            0   3
         0  1   4
         1  5   8
         2  9  12

In [23]: df.dropna(axis='columns')
Out[23]:
            0   3
         0  1   4
         1  5   8
         2  9  12
```

关于 dropna()的完整参数列表及其含义，应该用“pd.dropna?”方法来查看文档，这个操作请读者自行完成，并认真阅读文档。

```
In [24]: df.fillna(-999)
Out[24]:
            0      1       2   3
         0  1    2.0  -999.0   4
         1  5 -999.0     7.0   8
         2  9   10.0    11.0  12

In [25]: df
Out[25]:
            0      1       2   3
         0  1    2.0     NaN   4
         1  5    NaN     7.0   8
         2  9   10.0    11.0  12
```

fillna()的作用是用某个值把数据中的 NaN 替换掉，如 In[24]的操作。这种替换不仅仅是替换为某个固定的值，还可以像下面这样做。

```
In [26]: s.fillna(method='ffill')    #使用 In[12]修改后的对象 s。根据前面的值替换 NaN。
Out[26]: 0    NaN    #因为前面没有值，所以不变
         1    NaN    #前面是 NaN，此处依然保持
         2    3.0
         3    3.0     #用前面的值替换此处原 NaN
         4    7.0
         5    9.0
         dtype: float64

In [27]: s.fillna(method='bfill')    #根据后面的值替换 NaN
Out[27]: 0    3.0    #用后面的值替换此处原 NaN
         1    3.0    #用后面的值替换此处原 NaN（从下向上看）
         2    3.0
         3    7.0
         4    7.0    #用后面的值替换此处原 NaN
         5    9.0
         dtype: float64
```

所有上述替换操作，都会新生成一个对象，没有修改原有的对象 s。如果是 DataFrame 对象，则要规定好方向，即沿着哪个轴的方向来确定前后替换。

```
In [28]: df.fillna(method='ffill', axis=1)
Out[28]:
      0     1     2     3
0  1.0   2.0   2.0   4.0
1  5.0   5.0   7.0   8.0
2  9.0  10.0  11.0  12.0
```

还要说明，上述仅仅是 fillna()的简单操作，完整的参数说明请用“pd.fillna?”方法查看文档。

NaN 不仅会在创建的数据中生产，在数据运算等其他操作中也会产生，所以它将伴随我们后续的很多操作。

2.6　规整数据

在数据处理项目中，对数据进行各种规整操作（这里暂时不用“清洗数据”的说法，因为“清洗数据”或“数据清洗”与本节所言还不是完全一致）是避免不了的，其工作量很大。在第 1 章中，曾经有一节专门讨论对数组的各种组合、分割操作，那些就是用来规整数据的一些方法。本节介绍 Pandas 中的一些方法，这些方法能够帮助我们更便捷地完成对数据的规整。

1．轴向连接

在 NumPy 中，有一个名为 np.concatenate()的函数，这个函数通过对 axis 参数进行赋值，实现沿着所设置的轴方向连接数组。Pandas 也提供了一个类似的函数，不过其参数更丰富。

```
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, copy=True)
```

为了理解 pd.concat()的应用，我们首先对其部分常用参数进行说明（具体内容如表 2-6-1 所示），然后用示例进行相关说明。

表 2-6-1　pd.concat()部分参数列表

参　数	类　型	说　明
objs	序列或 Series、DataFrame、Panel、字典等对象	参与连接的对象，是必需的，不能省略
axis	0 或 1	默认为 0。指定沿着哪个轴的方向连接
join	inner 或 outer	默认为 outer。其他轴上的索引是按照 outer（并集）还是 inner（交集）连接
join_axes	索引组成的列表	指定其他 *n*-1 条轴索引，不执行交集/并集操作。比如对于 DataFrame 对象，当 axis=0 时，若 join_axes 是列表，则该列表的值就是 DataFrame 的列标签
ignore_index	布尔型	默认为 False。如果为 True，就摒弃连接轴的索引，代之以 0, ..., *n* – 1 的整数，这是一种新的索引
keys	序列	默认为 None。用于设置多级索引
levels	序列组成的列表	默认为 None。指定用于多级索引的标签
names	列表	默认为 None。多级索引的名称
verify_integrity	布尔型	默认为 False。如果为 True，当检查到连接的轴有重复时，会发起异常

从最简单的连接操作开始，一步一步地趋近于对 pd.concat()的全面理解。

```
In [1]: import numpy as np
        import pandas as pd
        s1 = pd.Series([100, 200, 300], index=['a', 'b', 'c'])
        s2 = pd.Series([400, 500, 600], index=['d', 'e', 'f'])
        pd.concat([s1, s2])
Out[1]: a    100
        b    200
        c    300
        d    400
        e    500
        f    600
        dtype: int64
```

pd.concat([s1, s2])仅仅给 objs 参数赋值，即连接两个 Series 对象，其他参数都采用默认值，默认连接方向是 0 轴方向，也就是将两个 Series 对象堆叠起来，如图 2-6-1 所示。

s1	
a	100
b	200
c	300

s2	
d	400
e	500
f	600

result	
a	100
b	200
c	300
d	400
e	500
f	600

图 2-6-1　In[1]的操作示意图

如果连接对象是 DataFrame 对象，应该如何？

```
In [2]: df1 = pd.DataFrame([[110, 120, 130], [210, 220, 230], [310, 320, 330]])
        df2 = pd.DataFrame([[11, 12, 13], [21, 22, 23], [31, 32, 33]])
        pd.concat([df1, df2])
Out[2]:
            0    1    2
        0  110  120  130
        1  210  220  230
        2  310  320  330
        0   11   12   13
        1   21   22   23
        2   31   32   33
```

In[1]和 In[2]中的连接操作都是沿 0 轴方向的，而且它们所连接的数据结构是一样的。如果对象的结构不同，比如列的数量有差异了，会如何呢？比较极端的例子是将 df1 和 s1 连接起来，一个是二维的，另一个是一维的，看看效果如何。

```
In [3]: pd.concat([df1, s1])
Out[3]:
            0      1      2
        0  110  120.0  130.0
        1  210  220.0  230.0
        2  310  320.0  330.0
        a  100    NaN    NaN
        b  200    NaN    NaN
        c  300    NaN    NaN
```

Series 对象 s1 只有一列（只有 0 轴），Pandas 自动将其数据与 df1[0]的列数据对齐，而 df1 的另外两列下面，自动补充 NaN。还要注意一点，在结果数据中，有 NaN 的列的数据类型都变成了浮点数，即使原来 df1 中是整数。这是因为 NaN 是浮点数，Pandas 自动做了类型转换。类似的情况在后面还会看到。

仔细观察上述演示连接后得到的结果，其索引也是顺序连接的。为了在结果中也能明显区分所连接对象的来源，可以使用多级索引，所需要做的就是指定 keys 参数的值。

```
In [4]: pd.concat([df1, s1], keys=['df1', 's1'])
Out[4]:
              0      1      2
       df1  0  110  120.0  130.0
            1  210  220.0  230.0
            2  310  320.0  330.0
       s1   a  100    NaN    NaN
            b  200    NaN    NaN
            c  300    NaN    NaN
```

要实现沿着 1 轴方向连接，设定参数 axis=1 即可。

```
In [5]: pd.concat([df1, df2], axis=1)
Out[5]:
           0    1    2   0   1   2
       0  110  120  130  11  12  13
       1  210  220  230  21  22  23
       2  310  320  330  31  32  33
```

df1 和 df2 也很巧合，正好其索引都采用了位置索引，即 0、1、2，如果不是这样，像下面这样呢？

```
In [6]: df1.index = ['a', 'c', 'd']
        df1
Out[6]:
           0    1    2
       a  110  120  130
       c  210  220  230
       d  310  320  330

In [7]: df2.index = ['b', 'c', 'd']
        df2
Out[7]:
          0   1   2
       b  11  12  13
       c  21  22  23
       d  31  32  33
```

仔细观察，df1 和 df2 的索引不完全对应，如果还是用类似 In[5]的操作会怎样呢？如图 2-6-2 所示为 In[8]的操作示意图。

```
In [8]: pd.concat([df1, df2], axis=1)    #与 pd.concat([df1, df2], axis=1, join='outer')
一样
Out[8]:
           0      1      2    0    1    2
```

```
a  110.0  120.0  130.0   NaN   NaN   NaN
b    NaN    NaN    NaN  11.0  12.0  13.0
c  210.0  220.0  230.0  21.0  22.0  23.0
d  310.0  320.0  330.0  31.0  32.0  33.0
```

df1

	0	1	2
a	110	120	130
c	210	220	230
d	310	320	330

df2

	0	1	2
b	11	12	13
c	21	22	23
d	31	32	33

result

	0	1	2	0	1	2
a	110.0	120.0	130.0	NaN	NaN	NaN
b	NaN	NaN	NaN	11.0	12.0	13.0
c	210.0	220.0	320.0	21.0	22.0	23.0
d	310.0	320.0	330.0	31.0	32.0	33.0

图 2-6-2　In[8]的操作示意图

当设置了 axis=1 后，df1 和 df2 的连接方向就改为沿着 1 轴方向连接了。注意观察结果，首先看 1 轴方向，也就是列标签，依次将 df1、df2 的列标签分别排列；再看 0 轴方向，即结果的索引，是原来两个数据的索引的并集，这是因为我们使用了默认的 join='outer'参数设置。如果设置为 join='inner'，结果如何？会得到索引的交集（如 In[9]所示）。最后看数据，Out[8]结果数据的索引有数据缺失，Pandas 自动补充 NaN。

```
In [9]: pd.concat([df1, df2], axis=1, join='inner')
Out[9]:
    0   1   2   0   1   2
c  210  220  230  21  22  23
d  310  320  330  31  32  33
```

在沿 1 轴方向连接数据的时候，同样可以设置列的多级索引，如下所示。

```
In [10]: df1.columns = ['one', 'two', 'three']
         df2.columns = ['one', 'three', 'five']
         pd.concat([df1, df2], axis=1, keys=['level1', 'level2'], names=['LEVEL', 'ROW'])
Out[10]:
         LEVEL level1                level2
         ROW   one     two     three   one   three  five
         a     110.0   120.0   130.0   NaN   NaN    NaN
         b     NaN     NaN     NaN     11.0  12.0   13.0
         c     210.0   220.0   230.0   21.0  22.0   23.0
         d     310.0   320.0   330.0   31.0  32.0   33.0
```

下面是另外一种设置列的多级索引的方式。

```
In [11]: pd.concat({"level1": df1, "level2": df2}, axis=1, names=['LEVEL', 'ROW'])
Out[11]:
         LEVEL level1                level2
         ROW   one     two     three   one   three  five
         a     110.0   120.0   130.0   NaN   NaN    NaN
         b     NaN     NaN     NaN     11.0  12.0   13.0
         c     210.0   220.0   230.0   21.0  22.0   23.0
         d     310.0   320.0   330.0   31.0  32.0   33.0
```

In[10]和 In[11]的操作等效，区别在于 pd.concat()传给参数 objs 的值的类型不同。

在合并数据的时候，还可以使用 join_axes 参数，指明列标签。

```
In [12]: pd.concat([df1, df2], join_axes=[df1.columns])
Out[12]:
           one    two  three
        a  110   120.0    130
        c  210   220.0    230
        d  310   320.0    330
        b   11     NaN     12
        c   21     NaN     22
        d   31     NaN     32
```

以上演示了连接两个数据对象的操作，多于两个数据对象的情况亦然，只是在参数 objs 的列表中多了几个元素。

除使用 pd.concat()方法能够完成轴向连接外，对于 Series 对象和 DataFrame 对象而言，还有一个实例方法 append()，只不过它不实现在 axis=1 轴方向的连接，这个方法所完成的操作与 pd.concat()的 axis=0 轴方向连接相当。

```
In [13]: df1.append(df2)
Out[13]:
           five   one  three   two
        a   NaN  110    130  120.0
        c   NaN  210    230  220.0
        d   NaN  310    330  320.0
        b  13.0   11     12    NaN
        c  23.0   21     22    NaN
        d  33.0   31     32    NaN

In [14]: df1.append(df2, ignore_index=True)
Out[14]:
           five   one  three   two
        0   NaN  110    130  120.0
        1   NaN  210    230  220.0
        2   NaN  310    330  320.0
        3  13.0   11     12    NaN
        4  23.0   21     22    NaN
        5  33.0   31     32    NaN
```

In[14]中用参数 ignore_index=True 忽略了原有的索引，对结果重新建立位置索引。还要请读者注意，Pandas 实例对象的 append()方法和在 Python 中所学的列表的 append()方法有很大的区别。在 In[13]中虽然操作了 df1.append(df2)，但 df1 的值并没有改变，Out[13]的结果是新生成的对象，列表中的 append()则是对其进行原地修改。

2．合并数据

在 Pandas 中还有一个名为 merge()的函数，它的中文意思是“合并”，那么它跟 pd.concat()有什么区别？看示例。

```
In [15]: gdp1 = pd.DataFrame({"city": ["shanghai", "guangzhou", "shenzhen", "chognqing"],
                              "number":[27466.2, 19610.9, 19492.6, 17558.8]})
         gdp2 = pd.DataFrame({"city": ["shanghai", "beijing", "shenzhen", "suzhou"],
                              "number":[27466.2, 24899.3, 19492.6, 15475.1]})
```

```
In [16]: gdp1
Out[16]:
              city   number
         0   shanghai   27466.2
         1   guangzhou  19610.9
         2   shenzhen   19492.6
         3   chognqing  17558.8

In [17]: gdp2
Out[17]:
               city   number
         0   shanghai  27466.2
         1   beijing   24899.3
         2   shenzhen  19492.6
         3   suzhou    15475.1

In [18]: pd.merge(gdp1, gdp2)
Out[18]:
               city   number
         0  shanghai  27466.2
         1  shenzhen  19492.6

In [19]: pd.concat([gdp1, gdp2], join='inner', axis=1)
Out[19]:
               city    number       city    number
         0   shanghai   27466.2   shanghai  27466.2
         1   guangzhou  19610.9   beijing   24899.3
         2   shenzhen   19492.6   shenzhen  19492.6
         3   chognqing  17558.8   suzhou    15475.1
```

In[18]中的 pd.merger(gdp1, gdp2)操作结果是对 gdp1 和 gdp2 两个 DataFrame 对象取并集（合并），而 In[19]的 pd.concat([gdp1, gdp2], join='inner', axis=1)是沿着 axis=1 轴方向将二者连接。请读者对照结果，体会 merge()和 concat()的区别。为了更严谨，还是要看一下 pd.merge()的文档内容（在 Jupyter 中输入“pd.merge?”查看文档）。

```
Signature: pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True,
indicator=False)

Docstring:
Merge DataFrame objects by performing a database-style join operation by columns or
indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if
joining indexes on indexes or indexes on a column or columns, the index will be passed
on.
#省略后面的参数解释，对部分参数的说明见下文。
```

从文档中对 pd.merge()的描述可以看出，此函数是针对 DataFrame 对象而言的，并且对其实施的是“database-style join”（姑且翻译为“数据库风格合并”）。何谓“数据库风格”？数据库表中的数据可以看作是二维数组，或者是 DataFrame 对象（横着的字段名称就是 DataFrame 中

的列标签，竖着的默认记录 id 就是 DataFrame 的位置索引）。如果将不同数据库表中的数据合并为一张表，则要指定一个键（key，是数据库表中的一个字段，从 DataFrame 的角度看，就是某一个或者几个列标签），然后根据这个键将不同来源的数据合并到一起，这就是所谓的“数据库风格合并”。根据这个解释再反观 In[18]和 In[19]的操作及其结果。

不论读者现在英语阅读能力如何，笔者都建议把上述英文说明认真阅读完毕，对理解 pd.merge()的应用大有裨益，然后结合表 2-6-2 中对部分参数的解释，就能够基本了解这个函数的功能了。

表 2-6-2　pd.merge()部分参数表

参　数	类　型	说　明
left/right	DataFrame	不能省略。进行合并的两个 DataFrame 对象
how	字符串 'left', 'right', 'outer', 'inner'	默认为'inner'。 • left/right：仅以 left/right 的 DataFrame 对象中所指定的列为键、以并集（outer）的方式合并数据。 • outer：两个 DataFrame 对象指定键的并集。 • inner：两个 DataFrame 对象指定键的交集
on	字符串或列表	默认为 None。用于合并的字段名称（列标签）。一般是两个 DataFrame 所共有的列。如果 on=None，且未指定合并的键，则以两个 DataFrame 对象列标签的交集为合并的键
left_on/right_on	字符串或列表、类数组	默认为 None。从 left/right 所引用的 DataFrame 对象中选择记录（列标签）作为数据合并的键
left_index/right_index	布尔型	默认为 False。如果为 True，则以 left/right 所引用对象的索引为键合并数据
suffixes	元组、列表类序列，长度为 2	默认为('_x', '_y')。左右两侧相同列名称的后缀
indicator	布尔型或字符串	默认为 False。如果为 True，在返回的 DataFrame 对象中会增加列标签“_merge”的列，注明每行数据的来源。如果是字符串，则此列标签为该字符串

在有了初步认识的基础上，再通过具体的代码体验一番。还是那句“纸上得来终觉浅，绝知此事要躬行”，在我们这里就是“看看文档还不行，绝知此事敲代码”（不押韵）——这个方法跟前面学习 pd.concat()的方法一样。

```
In [20]: left_df = pd.DataFrame({"grade":['a', 'b', 'b', 'a', 'c', 'c', 'd'],
                                 "ldata":range(7)})
         right_df = pd.DataFrame({"grade":['a', 'b', 'c'], "rdata":[30, 40, 50]})
         pd.merge(left_df, right_df)
Out[20]:
           grade  ldata  rdata
         0     a      0     30
         1     a      3     30
         2     b      1     40
         3     b      2     40
         4     c      4     50
         5     c      5     50

In [21]: pd.merge(left_df, right_df, on="grade", how='inner')
```

```
Out[21]:
   grade  ldata  rdata
0     a      0     30
1     a      3     30
2     b      1     40
3     b      2     40
4     c      4     50
5     c      5     50
```

仔细观察 left_df 和 right_df 两个变量所引用的 DataFrame 对象，它们都有名为“grade”的列标签，于是在 In[20]的操作中，就默认以此列为键（key）进行合并，还默认得到两个 DataFrame 对象的键的交集及其所在行数据，如 Out[20]所示。In[21]将默认的设置显化，Out[20]和 Out[21]所示的结果一样。图 2-6-3 显示了上述合并过程。

left

	grade	ldata
0	a	0
1	b	1
2	b	2
3	a	3
4	c	4
5	c	5
6	d	6

right

	grade	rdata
0	a	30
1	b	40
2	c	50

result

	grade	ldata	rdata
0	a	0	30
1	a	3	30
2	b	1	40
3	b	2	40
4	c	4	50
5	c	5	50

图 2-6-3　In[21]的操作示意图

能够被指定为键的列也可以是多个。比如下面的示例（请读者特别注意观察“并集”的特点）。

```
In [22]: ldf = pd.DataFrame({"key1":['K0', 'K0', 'K1', 'K2'],  'key2': ['K0', 'K1', 'K0',
                             'K1'],'A': ['A0', 'A1', 'A2', 'A3'],'B': ['B0', 'B1', 'B2', 'B3']})
         rdf = pd.DataFrame({"key1":['K0', 'K1', 'K1', 'K2'],  'key2': ['K0', 'K0', 'K0',
                             'K0'], 'C': ['C0', 'C1', 'C2', 'C3'], 'D': ['D0', 'D1', 'D2', 'D3']})
         pd.merge(ldf, rdf, on=["key1", "key2"], how="inner")
Out[22]:
     A   B  key1  key2  C   D
0  A0  B0   K0    K0  C0  D0
1  A2  B2   K1    K0  C1  D1
2  A2  B2   K1    K0  C2  D2
```

在 In[22]的操作中，显示了 how 的默认参数，如果将其设置为其他参数，那么合并结果会如何？

```
In [23]: pd.merge(left_df, right_df, on="grade", how="left")
Out[23]:
   grade   ldata  rdata
0     a       0   30.0
1     b       1   40.0
2     b       2   40.0
3     a       3   30.0
4     c       4   50.0
5     c       5   50.0
6     d       6    NaN
```

In[23]中以 how="left"指定用 left_df 的 grade 列为键进行合并，此合并过程可以用图 2-6-4 进行说明。

left

	grade	ldata
0	a	0
1	b	1
2	b	2
3	a	3
44	c	4
5	c	5
6	d	6

right

	grade	rdata
0	a	30
1	b	40
2	c	50

result

	grade	ldata	rdata
0	a	0	30.0
1	b	1	40.0
2	b	2	40.0
3	a	3	30.0
4	c	4	50.0
5	c	5	50.0
6	d	6	NaN

图 2-6-4　In[23]的操作示意图

当然也可以用 how="right"，请读者自行测试。如果是 how="outer"，得到的结果应该是两个数据的 grade 列的并集，其结果与 Out[23]一样，因为 left_df['grade']是 right_def['grade']的超集。改用前面创建的 ldf 和 rdf 两个 DataFrame 对象，体会一下 how="outer"的效果，并对比 how="left"的结果。

```
In [24]: pd.merge(ldf, rdf, on=["key1", "key2"], how="outer")
Out[24]:
     A    B  key1 key2    C    D
0   A0   B0    K0   K0   C0   D0
1   A1   B1    K0   K1  NaN  NaN
2   A2   B2    K1   K0   C1   D1
3   A2   B2    K1   K0   C2   D2
4   A3   B3    K2   K1  NaN  NaN
5  NaN  NaN    K2   K0   C3   D3

In [25]: pd.merge(ldf, rdf, on=["key1", "key2"], how="left")
Out[25]:
    A   B  key1 key2    C    D
0  A0  B0    K0   K0   C0   D0
1  A1  B1    K0   K1  NaN  NaN
2  A2  B2    K1   K0   C1   D1
3  A2  B2    K1   K0   C2   D2
4  A3  B3    K2   K1  NaN  NaN
```

如果在 pd.merge()中启用 indicator=True，则会让输出结果更友好。

```
In [26]: pd.merge(ldf, rdf, on=["key1", "key2"], how="left", indicator=True)
Out[26]:
    A   B  key1  key2   C    D      _merge
0  A0  B0    K0    K0   C0   D0        both
1  A1  B1    K0    K1  NaN  NaN   left_only
2  A2  B2    K1    K0   C1   D1        both
3  A2  B2    K1    K0   C2   D2        both
4  A3  B3    K2    K1  NaN  NaN   left_only
```

在以上各例中，left 和 right 所引用的对象都有共同名称的列标签，并将其指定为合并所依据的键。还有一种情况，就是两个待合并的 DataFrame 对象没有同名的列标签，还能不能合并？新建一个 new_ldf 的 DataFrame 对象，在 ldf 所引用的对象基础上修改一下即可。

```
In [27]: new_ldf = pd.DataFrame({"jian1":['K0', 'K0', 'K1', 'K2'],  'jian2': ['K0', 'K1',
                                 'K0', 'K1'], 'A': ['A0', 'A1', 'A2', 'A3'],'B': ['B0',
```

```
                                'B1', 'B2', 'B3']})
        new_ldf
Out[27]:
            A   B  jian1   jian2
        0  A0  B0    K0    K0
        1  A1  B1    K0    K1
        2  A2  B2    K1    K0
        3  A3  B3    K2    K1

In [28]: rdf
Out[28]:
            C   D  key1  key2
        0  C0  D0   K0   K0
        1  C1  D1   K1   K0
        2  C2  D2   K1   K0
        3  C3  D3   K2   K0
```

下面将 new_ldf 和 rdf 两个变量所引用的 DataFrame 对象合并，注意它们没有相同的列标签。

```
In [29]: pd.merge(new_ldf, rdf, left_on=["jian1", "jian2"], right_on=["key1", "key2"],
                  how="left", indicator=True)
Out[29]:
            A   B  jian1  jian2    C   D   key1  key2    _merge
        0  A0  B0     K0     K0   C0   D0    K0   K0       both
        1  A1  B1     K0     K1  NaN  NaN  NaN  NaN  left_only
        2  A2  B2     K1     K0   C1   D1    K1   K0       both
        3  A2  B2     K1     K0   C2   D2    K1   K0       both
        4  A3  B3     K2     K1  NaN  NaN  NaN  NaN  left_only
```

为了能够更清晰地理解上述合并过程，还是用图来表示，如图 2-6-5 所示，请仔细观察。

new_ldf

	A	B	jian1	jian2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

rdf

	C	D	key1	key2
0	C0	D0	K0	K0
1	C1	D1	K1	K0
2	C2	D2	K1	K0
3	C3	D3	K2	K0

result

	A	B	jian1	jian2	C	D	key1	key2	_merge
0	A0	B0	K0	K0	C0	D0	K0	K0	both
1	A1	B1	K0	K1	NaN	NaN	NaN	NaN	left_only
2	A2	B2	K1	K0	C1	D1	K1	K0	both
3	A2	B2	K1	K0	C2	D2	K1	K0	both
4	A3	B3	K2	K1	NaN	NaN	NaN	NaN	left_only

图 2-6-5　In[29]的操作示意图

除能够指定列为合并数据的键外，还可以指定 DataFrame 对象的索引为合并数据的键。

```
In [30]: rdf_index = rdf.set_index("key1")  #将 In[22]中创建的 rdf 对象的列 key1 设置为索引
         rdf_index
Out[30]:
                C   D  key2
        key1
        K0     C0  D0   K0
        K1     C1  D1   K0
        K1     C2  D2   K0
```

```
        K2   C3  D3   K0

In [31]: ldf_index = new_ldf.set_index("jian1") #将 In[27]中 new_ldf 的列 jian1 设置为索引
         ldf_index
Out[31]:
               A   B  jian2
         jian1
         K0    A0  B0   K0
         K0    A1  B1   K1
         K1    A2  B2   K0
         K2    A3  B3   K1

In [32]: pd.merge(ldf_index, rdf_index, left_index=True, right_index=True)
Out[32]:
             A   B  jian2   C   D  key2
         K0  A0  B0    K0  C0  D0   K0
         K0  A1  B1    K1  C0  D0   K0
         K1  A2  B2    K0  C1  D1   K0
         K1  A2  B2    K0  C2  D2   K0
         K2  A3  B3    K1  C3  D3   K0
```

在了解了 pd.merge()的基本操作之后，关于合并数据的话题还没有结束。前面学习 pd.concat()的时候，提到了一个实现轴向连接的实例对象函数 append()，与之类似，在实现合并数据的操作中，也有一个 DataFrame 实例对象的函数 join()。

```
In [33]: ldf_index.join(rdf_index)
Out[33]:
             A   B  jian2   C   D  key2
         K0  A0  B0    K0  C0  D0   K0
         K0  A1  B1    K1  C0  D0   K0
         K1  A2  B2    K0  C1  D1   K0
         K1  A2  B2    K0  C2  D2   K0
         K2  A3  B3    K1  C3  D3   K0
```

在默认情况下，df.join()是以“索引为键”与另外一个 DataFrame 实例对象合并的，也可以通过参数 on 指定以实例对象的“列为键”进行合并。更完整的使用方法请通过阅读官方文档理解。下面是 join()函数的完整参数，每个参数的含义与 pd.merge()中的参数含义一致，读者可以参照学习。

```
ldf_index.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)
```

以上我们合并了索引都是位置索引的两个 DataFrame 对象，也合并了索引都是标签索引的两个 DataFrame 对象。根据排列组合的原理，应该再学习一个是标签索引、另一个是位置索引的 DataFrame 对象的合并。

```
In [34]: rdf    #rdf 是 In[22]中所创建的 DataFrame 对象
Out[34]:
            C   D  key1  key2
         0  C0  D0   K0    K0
         1  C1  D1   K1    K0
         2  C2  D2   K1    K0
         3  C3  D3   K2    K0
```

```
In [35]: pd.merge(ldf_index, rdf, left_index=True, right_on="key1")
Out[35]:
            A   B  jian2   C   D   key1  key2
         0  A0  B0     K0  C0  D0     K0    K0
         0  A1  B1     K1  C0  D0     K0    K0
         1  A2  B2     K0  C1  D1     K1    K0
         2  A2  B2     K0  C2  D2     K1    K0
         3  A3  B3     K1  C3  D3     K2    K0
```

还有多级索引，也不能忘记。不过，笔者就不再赘述了，请读者自行测试一下多级索引的 DataFrame 对象合并，基本方法同前。

下面要做一个练习（本练习流程参考了 *Python Data Science Handbook* 一书第 154~158 页的部分代码）。

读者可以在本书的代码仓库中下载所需的数据文件，也可以直接从数据源网站下载。

```
$ wget https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-
population.csv
$ wget https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-areas.csv
$ wget https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-
abbrevs .csv
```

数据文件下载之后，用我们已经熟悉的方式打开。

```
In [36]: pop = pd.read_csv("/home/qiwsir/Documents/data_analysis/state-population.csv")
         areas = pd.read_csv("/home/qiwsir/Documents/data_analysis/state-areas.csv")
         abbrevs = pd.read_csv("/home/qiwsir/Documents/data_analysis/state-abbrevs.
         csv")pop.head()
Out[36]:
            state/region      ages    year   population
         0            AL   under18    2012    1117489.0
         1            AL     total    2012    4817528.0
         2            AL   under18    2010    1130966.0
         3            AL     total    2010    4785570.0
         4            AL   under18    2011    1125763.0

In [37]: areas.head()
Out[37]:
                     state  area (sq. mi)
         0         Alabama          52423
         1          Alaska         656425
         2         Arizona         114006
         3        Arkansas          53182
         4      California         163707

In [38]: abbrevs.head()
Out[38]:
                     state     abbreviation
         0         Alabama             AL
         1          Alaska             AK
         2         Arizona             AZ
         3        Arkansas             AR
```

```
4      California              CA
```

观察上面示例的输出结果，pop['state/region']列和 abbrevs[abbreviation]列都是美国各州名字的简称。下面就分别以这两列为键将两个数据以并集合并，目的是为 pop 中的每个简称都配上全称。因为在 abbrevs 中也有一列简称，所以还要删除该列。根据这些要求，操作如下。

```
In [39]: pop_merged = pd.merge(pop, abbrevs, left_on="state/region",
                               right_on="abbreviation", how="outer")
         pop_merged = pop_merged.drop('abbreviation', axis=1)
         pop_merged.head()
Out[39]:
            state/region      ages    year   population     state
         0            AL   under18    2012    1117489.0   Alabama
         1            AL     total    2012    4817528.0   Alabama
         2            AL   under18    2010    1130966.0   Alabama
         3            AL     total    2010    4785570.0   Alabama
         4            AL   under18    2011    1125763.0   Alabama
```

合并数据后，从规整数据的角度看，要对这些数据做深入的了解，比如检验数据记录中是否有空数据的情况出现。

```
In [40]: pop_merged.isnull().any()
Out[40]: state/region    False
         ages            False
         year            False
         population       True
         state            True
         dtype: bool
```

仔细看一下 population 列为空的那些记录。

```
In [41]: pop_merged[pop_merged['population'].isnull()].head()
Out[41]:
             state/region      ages    year  population   state
         2448          PR   under18    1990         NaN     NaN
         2449          PR     total    1990         NaN     NaN
         2450          PR     total    1991         NaN     NaN
         2451          PR   under18    1991         NaN     NaN
         2452          PR     total    1993         NaN     NaN
```

这里显示出某些记录没有 population 和 state 的数据。但是，由于我们看到的是有限个数的数据，不知道是否还有其他缺少上述数据的记录呢？虽然这里显示的都是 state/region 列的值为 PR 的记录。

```
In [43]: pop_merged.loc[pop_merged['state'].isnull(), 'state/region'].unique()
Out[43]: array(['PR', 'USA'], dtype=object)
```

通过 In[43]我们发现，在 pop_merged['state']是 NaN 的所有记录中，pop_merged['state/region']的值是'PR'和'USA'。这个发现很重要，先把这两个简称所对应的全称补上。

```
In [44]: pop_merged.loc[pop_merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
         pop_merged.loc[pop_merged['state/region'] == 'USA', 'state'] = 'United States'
         pop_merged.isnull().any()
```

```
Out[44]: state/region     False
         ages             False
         year             False
         population       True
         state            False
         dtype: bool
```

从 Out[44]的显示结果中可以得知，state 列没有缺失的数据了，现在只剩下 population 列了，暂时不管它，况且在实际项目中，这一列有缺失数据是正常的。下面要做的是把另外一个数据 areas（In[36]所创建的对象）中各个州的面积数据也合并到 pop_merged 中，这两个 DataFrame 对象中都有 state 列。

```
In [45]: result = pd.merge(pop_merged, areas, on='state', how='left')
         result.head()
Out[45]:
           state/region    ages  year   population     state area (sq. mi)
         0           AL  under18  2012    1117489.0   Alabama       52423.0
         1           AL   total  2012    4817528.0    labama       52423.0
         2           AL  under18  2010    1130966.0   Alabama       52423.0
         3           AL   total  2010    4785570.0   Alabama       52423.0
         4           AL  under18  2011    1125763.0   Alabama       52423.0
```

同样，再检查一下，新合并的数据是不是完整。

```
In [46]: result.isnull().any()
Out[46]: state/region     False
         ages             False
         year             False
         population       True
         state            False
         area (sq. mi)    True
         dtype: bool
```

果然，新合并进来的 area (sq. mi)列的数据不是很完整，还是使用类似 In[43]的方式，找出缺失数据的行有什么特征。

```
In [47]: result.loc[result['area (sq. mi)'].isnull(), 'state'].unique()
Out[47]: array(['United States'], dtype=object)

In [48]: result['state'][result['area (sq. mi)'].isnull()].unique()    #与 In[47]等效的操作
Out[48]: array(['United States'], dtype=object)
```

看到这个结果，应该理解了，areas 是各州的面积，没有全美的总面积。要解决这个问题，一个思路是把这个数据补上，还有另外一个思路就是把相关行删除。所以，处理问题的思路有两个：一个是解决问题，另外一个是解决提出问题的人。

```
In [49]: result.dropna(inplace=True)
         result.isnull().any()
Out[49]: state/region    False
         ages            False
         year            False
         population      False
         state           False
```

```
area (sq. mi)    False
dtype: bool
```

“消灭”之，一片祥和！

注意观察，In[49]中使用了参数 inplace=True，意思是让删除 NaN 的操作实现“原地修改”，只返回 None 或者理解为无返回值，这如同列表中的原地修改（在 Pandas 中若要进行原地修改，需要使用 inplace=True 参数）。

接下来就可以对 result 数据进行其他操作了，比如根据条件查询某些数据（后面会有相应函数的详细介绍）。

```
In [50]: d2012 = result.query("year==2012 & ages=='total'")
         d2012.head()
Out[50]:
           state/region   ages   year   population       state area (sq. mi)
         1           AL   total  2012    4817528.0     Alabama       52423.0
         95          AK   total  2012     730307.0      Alaska      656425.0
         97          AZ   total  2012    6551149.0     Arizona      114006.0
         191         AR   total  2012    2949828.0    Arkansas       53182.0
         193         CA   total  2012   37999878.0  California      163707.0
```

顺便计算一下每个州的人口密度吧。

```
In [51]: d2012 = result.query("year==2012 & ages=='total'")
         d2012.set_index('state', inplace=True)
         density = d2012['population'] / d2012['area (sq. mi)']
         density.head()
Out[51]: state
         Alabama       91.897221
         Alaska         1.112552
         Arizona       57.463195
         Arkansas      55.466662
         California   232.121278
         dtype: float64
```

显然 density 没有排序，需要排序才能看出哪里人口稠密、哪里人烟稀少。

```
In [52]: density.sort_values(ascending=False, inplace=True)
         density.head()
Out[52]: state
         District of Columbia   9315.102941
         Puerto Rico            1038.846373
         New Jersey             1016.710502
         Rhode Island            679.808414
         Connecticut             647.865260
         dtype: float64

In [53]: density.tail()
Out[53]: state
         South Dakota   10.814785
         North Dakota    9.919453
         Montana         6.837955
         Wyoming         5.894886
```

```
Alaska          1.112552
dtype: float64
```

在这个示例中，综合运用了多种关于数据的操作，有的知识需要在后续详解。读者或许练习之后还有一点疑惑，不要紧，权当对数据规整有了初步的了解。

3．组合数据

仅用 pd.concat()和 pd.merge()只是实现了“无选择”规整，还有另外一种可能，就是根据某个条件分别从不同的对象中选择数据，组成一个新的数据对象。打个比方，梁山好汉是一组数据（记作 L），瓦岗众英雄组成另外一组数据对象（记作 W），按照某个条件从 L 和 W 中选出若干个英雄好汉，组成一个新的数据对象。在数据分析中，这样的操作经常出现。

```
In [54]: a = pd.Series([np.nan, 2, 4, np.nan, 8, 10])
         b = pd.Series([1, 2, np.nan, np.nan, 5, np.nan])
         np.where(pd.isnull(b), a, b)
Out[54]: array([  1.,   2.,   4.,  nan,   5.,  10.])
```

In[54]中使用了 np.where()函数，根据条件对 a 和 b 中的数据元素进行筛选，最终得到根据条件选择出来的新的数据——这个数据不是合并和连接而成的，而是分别从 a 和 b 中取出部分若干元素新生成的 Series 对象，我们姑且把这种方式称为组合。

这是 NumPy 中提供的方法，Pandas 中有解法吗？有。

```
In [55]: a.combine_first?
Signature: a.combine_first(other)
Docstring:
Combine Series values, choosing the calling Series's values first. Result index will be
the union of the two indexes
```

仔细观察 In[55]获取帮助文档的方式，显然 combine_first 不是 Pandas 的类方法，而是一个实例方法，所以使用“a.combine_first?”方法来查看文档。

根据文档说明（请读者仔细阅读），来完成前述操作。

```
In [56]: b.combine_first(a)
Out[56]: 0    1.0
         1    2.0
         2    4.0
         3    NaN
         4    5.0
         5   10.0
         dtype: float64
```

比较 In[56]和 np.where(pd.isnull(b), a, b)的操作结果，还是有区别的（类型不同），但元素一致，实现了同样的目标。

DataFrame 类的实例对象也有 combine_first()方法。

```
In [57]: df1 = pd.DataFrame({"one":[11, 12, np.nan, 13, np.nan], "two":[np.nan, 22, 23,
                             np.nan, 24]})
         df2 = pd.DataFrame({"one":np.arange(6), "two":np.linspace(10, 100, 6),
                             "three":np.logspace(2, 3, 6)})
In [58]: df1
Out[58]:
```

```
    one   two
0  11.0   NaN
1  12.0  22.0
2   NaN  23.0
3  13.0   NaN
4   NaN  24.0
```

```
In [59]: df2
Out[59]:
   one       three    two
0    0   100.000000   10.0
1    1   158.489319   28.0
2    2   251.188643   46.0
3    3   398.107171   64.0
4    4   630.957344   82.0
5    5  1000.000000  100.0
```

比较 df1 和 df2 两个变量所引用的 DataFrame 对象，df1 中有一些缺失数据而 df2 中没有，且 df2 的行索引和列索引都比 df1 多。我们的需求是从 df2 中选择数据补充 df1，即缺失数据由 df2 补充，并且把缺少的行、列也从 df2 中取出来补充。

```
In [60]: df1.combine_first(df2)
Out[60]:
    one       three    two
0  11.0   100.000000   10.0
1  12.0   158.489319   22.0
2   2.0   251.188643   23.0
3  13.0   398.107171   64.0
4   4.0   630.957344   24.0
5   5.0  1000.000000  100.0
```

In[60]的操作实现了补充数据的目的。当然，返回的是一个新对象。

轴向连接、合并数据和组合数据，都是对多个数据源进行操作而得到一个需要的数据。此外，在规整数据的要求中，我们还会遇到对一个数据进行不同角度的转换。比如前面对美国各州人口、面积数据进行规整时，就遇到去除缺失数据的操作。所以，对单一数据的各种操作也不能少。

4．数据转换

无论是从查看还是从计算的角度看，都有可能对数据进行各种变换，比较常见的变换方式有行和列的转换（常用的是二维数据表）、去重、替换、检测异常等，这些都是清洗数据时会用到的基本操作。

（1）行列转换

对于行列的转换，我们在前面已经使用过了，这里权当复习，请看下面的示例。

```
In [61]: data = pd.DataFrame(np.arange(10).reshape(2, 5),
                             index=['one', 'two'], columns=['a', 'b', 'c', 'd', 'e'])
         data
Out[61]:
     a  b  c  d  e
one  0  1  2  3  4
```

```
         two  5  6  7  8  9

In [62]: data.stack()    #列转换为行
Out[62]: one  a    0
              b    1
              c    2
              d    3
              e    4
         two  a    5
              b    6
              c    7
              d    8
              e    9
         dtype: int64

In [63]: data.stack().unstack()    #行转换为列
Out[63]:
              a  b  c  d  e
         one  0  1  2  3  4
         two  5  6  7  8  9
```

stack()和 unstack()是 Series 对象或者 DataFrame 对象的实例方法。当然，对于多级索引的对象也适用。

提醒读者注意 stack()里面有一个参数 dropnan=True，请看下面的示例。

```
In [64]: data['e'] = np.nan
         data
Out[64]:
              a  b  c  d   e
         one  0  1  2  3 NaN
         two  5  6  7  8 NaN

In [65]: data.stack()
Out[65]: one  a    0.0
              b    1.0
              c    2.0
              d    3.0
         two  a    5.0
              b    6.0
              c    7.0
              d    8.0
         dtype: float64

In [66]: data.stack(dropna=False)
Out[66]: one  a    0.0
              b    1.0
              c    2.0
              d    3.0
              e    NaN
         two  a    5.0
              b    6.0
```

```
        c    7.0
        d    8.0
        e    NaN
    dtype: float64
```

因为默认 dropna=True，所以 In[65]的转换结果中就没有了 NaN；如果要保留，就要如同 In[66]那样操作。

（2）透视表

在数据库表中，字段名称都是固定的，并且一行一行地保存记录，如图 2-6-6 所示。

	class	subject	numbers
0	class1	physics	28
1	class2	python	30
2	class1	math	20
3	class2	physics	80

图 2-6-6　数据库表

“class”“subject”“numbers”就是字段名称，下面是一条一条的记录。按照这种结构，如果还有数据，可以继续增加。但是，对于这个数据表，仅能完成数据的一般记录。如果要显示 class 记录中每个班级（如 class1、class2）各自的 subject 和 numbers 值，则必须经过变换。在 Web 开发中，常常会遇到这种情况（关于 Web 开发，请阅读《跟老齐学 Python：Django 实战》）。

在电子表格中，有一种叫做“透视表”的操作，可以根据“class”数据再归类。在 Pandas 里面，也有一个“透视表”的函数，用来实现此功能。

```
In [67]: students = pd.DataFrame({"class":['class1', 'class2', 'class1', 'class2'],
                                  "subject":["physics", "python", "math", "physics"],
                                  "numbers":[28, 30, 20, 80]})
         students
Out[67]:
             class  numbers  subject
          0  class1       28  physics
          1  class2       30   python
          2  class1       20     math
          3  class2       80  physics

In [68]: students.pivot(index='class', columns='subject', values='numbers')
Out[68]:
         subject  math  physics  python
         class
         class1   20.0     28.0     NaN
         class2    NaN     80.0    30.0
```

pivot()是 DataFrame 对象的实例方法，在 In[68]的操作中已经很明确地指定了相关参数，从而将 DataFrame 对象的数据进行转换，具体转换的对应关系如图 2-6-7 所示。

	class	subject	numbers
0	class1	physics	28
1	class2	python	30
2	class1	math	20
3	class2	physics	80

subject	math	physics	python
class			
class1	20.0	28.0	NaN
class2	NaN	80.0	30.0

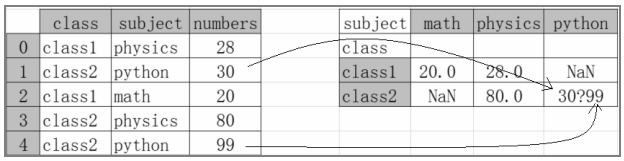
图 2-6-7　数据转换关系

在转换之后的 DataFrame 对象中，空缺部分用 NaN 填充。

当然，不是所有的 DataFrame 对象的数据都能够用 pivot()进行转换，比如下面的示例。

```
In [69]: students2 = pd.DataFrame({"class":['class1', 'class2', 'class1', 'class2',
                                            'class2'],
                                   "subject":["physics", "python",
                                   "math", "physics", "python"],
                                   "numbers":[28, 30, 20, 80, 99]})
         students2.pivot(index='class', columns='subject', values='numbers')
         …    #省略部分报错信息
         ValueError: Index contains duplicate entries, cannot reshape
```

仔细观察，会发现对于“class='class2',subject='python'”条件的数据，在转换后有 30 和 99 两个数据，如图 2-6-8 所示，让 Pandas 无所适从，所以干脆就罢工了。



	class	subject	numbers
0	class1	physics	28
1	class2	python	30
2	class1	math	20
3	class2	physics	80
4	class2	python	99

subject	math	physics	python
class			
class1	20.0	28.0	NaN
class2	NaN	80.0	30?99

图 2-6-8　转换后一个条件有两个数据的情况

难道 Pandas 就对此束手无策了吗？

Pandas 当然不会就此罢休，它提供了一个解决此问题的利器——即使 Pandas 不解决这个问题，读者也可以写一个名为 Sadnap 的库专门解决它，很多“大牛”都这么干。

```
In [70]: students2.pivot_table(index='class', columns='subject', values='numbers',
                               aggfunc=np.sum)
Out[70]:
         subject  math   physics  python
         class
         class1   20.0     28.0     NaN
         class2   NaN      80.0    129.0
```

pivot_table()是何方神圣？pivot()与 pivot_table()有什么区别？

要回答这些问题，最好看官方文档。首先，两者都是实例的方法，然后通过实例来查看两者的文档。

```
Signature: students2.pivot_table(values=None, index=None, columns=None, aggfunc='mean',
fill_value=None, margins=False, dropna=True, margins_name='All')
Docstring:
Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will
be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the
result DataFrame

Signature: students2.pivot(index=None, columns=None, values=None)
Docstring:
Reshape data (produce a "pivot" table) based on column values. Uses unique values from
index / columns to form axes of the resulting DataFrame.
```

如果认真阅读两个文档的说明，则可以很明显地看出它们的区别。这两个方法的参数的含义，请读者查阅帮助信息并自行阅读，其实从参数的名字也能知道其含义。

下面通过一个具体的示例综合运用透视表及以往学习过的内容。笔者提供了一份历年人口出生数据的 CSV 文档，可以在本书代码仓库中下载。

```
In [71]: cnpop = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter02/ cnpop.csv",
                             encoding='utf-8')
         cnpop.head()
Out[71]:
            year  birth_rate  death_rate  growth_rate
         0  1978       18.25        6.25        12.00
         1  1980       18.21        6.34        11.87
         2  1981       20.91        6.36        14.55
         3  1982       22.28        6.60        15.68
         4  1983       20.19        6.90        13.29
```

以 10 年为一个年代，再增加一个列标签，标记每行记录所处的年代。

```
In [72]: cnpop['decade'] = cnpop['year'] // 10 * 10
         cnpop.head()
Out[72]:
            year  birth_rate  death_rate  growth_rate  decade
         0  1978       18.25        6.25        12.00    1970
         1  1980       18.21        6.34        11.87    1980
         2  1981       20.91        6.36        14.55    1980
         3  1982       22.28        6.60        15.68    1980
         4  1983       20.19        6.90        13.29    1980
```

接下来我们使用 cnpop.piovt_table()函数，统计每个年代的人口出生率、死亡率和自然增长率的平均值。

```
In [73]: cnpop.pivot_table(index='decade', aggfunc='mean',
                           values=['birth_rate', 'death_rate', 'growth_rate'])
Out[73]:
                 birth_rate  death_rate   growth_rate
         decade
         1970    18.250000    6.250000    12.000000
         1980    21.224000    6.656000    14.568000
         1990    17.572000    6.574000    10.998000
         2000    12.565000    6.650000     5.915000
         2010    11.976667    7.133333     4.843333
```

观察上述统计结果，就不难理解现在的“二孩”政策了——科学决策需要数据的支持，所以，数据分析真的是“广阔天地，大有作为”。

2.7　分组运算

在数据分析中，有些基本的统计量经常需要计算，比如求和、求平均值、求中位数、求最大值和最小值等，在统计学中这些量被称为“集中量”。NumPy 中也有相应的函数来完成这些统计工作，因为 Pandas 对 NumPy 有接口，所以在 NumPy 中学习过的函数，对 Pandas 数据而言，也都能使用。

```
In [1]: import numpy as np
        import pandas as pd
```

```
In [2]: s = pd.Series(np.arange(5))
        print("sum = ", np.sum(s))
        print("mean = ", np.mean(s))
        sum =  10
        mean =  2.0
```

在 NumPy 中学过的知识都没有浪费。

此外，Pandas 对象本身也有计算这些集中量的方法。当然，既然是 Pandas 对象的方法，必然有自己独特的地方，这些独特的地方是 NumPy 中的方法无法替代的。比如本节将要学习的分组运算，通过其特有的方法实现了 NumPy 中不能直接实现的运算。

1．简单的统计运算

对于 Series 对象，计算各个集中量比较简单，与 In[2]所演示的用数组的方法类似，所以此处从略，读者可以自行测试。下面主要演示 DataFrame 对象的集中量函数的使用（类似 NumPy 中的使用方法）。

```
In [3]: gdp = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter02/gdp-
                          population.csv")
        city_gdp = gdp.set_index('City_Name')    #将“City_Name”列设为标签索引
        city_gdp
Out[3]:
                            GDP  Population
        City_Name
        SHANGHAI       27466.15     2419.70
        BEIJING        24899.30     2172.90
        GUANGZHOU      19610.90     1350.11
        SHENZHEN       19492.60     1137.87
        TIANJIN        17885.39     1562.12
        CHONGQING      17558.76     3016.55
        SUZHOU         15475.09     1375.00
        CHENGDU        12170.20     1591.76
```

city_gdp 是一个 DataFrame 对象，这个对象的方法中有相关集中量统计的函数，注意不是 Pandas 的类方法，是实例方法。

```
In [4]: city_gdp.sum()    #求和
Out[4]: GDP          154558.39
        Population    14626.01
        dtype: float64

In [5]: city_gdp.mean()    #平均值
Out[5]: GDP          19319.79875
        Population    1828.25125
        dtype: float64

In [6]: city_gdp.median()    #中位数
Out[6]: GDP          18688.995
        Population    1576.940
        dtype: float64
```

默认情况下，将沿着 0 轴方向对各列的数据进行统计，这是与数组不同的。请根据下述提示回忆数组的统计。

```
In [7]: b = np.arange(9).reshape(3,3)
        b.mean(), b.mean(axis=0)
Out[7]: (4.0, array([ 3.,  4.,  5.]))
```

比较后不难发现，因为 DataFrame 对象在设计上更接近于数据库表或者电子表格中的数据，所以其实例方法在没有声明轴的时候，默认按照各自的列进行统计。这符合通常的业务需求，也是 Pandas 相对 NumPy 在某领域应用的便捷之处。

当然，对 DataFrame 对象的方法同样可以设置 axis=1，就是沿着 1 轴方向进行统计。

```
In [8]: city_gdp.mean(axis=1)
Out[8]: City_Name
        SHANGHAI       14942.925
        BEIJING        13536.100
        GUANGZHOU      10480.505
        SHENZHEN       10315.235
        TIANJIN         9723.755
        CHONGQING      10287.655
        SUZHOU          8425.045
        CHENGDU         6880.980
        dtype: float64
```

虽然对于 city_gdp 而言，沿着 1 轴方向计算平均值没有什么实在意义，但是从演示结果中就能理解 axis=1 的含义。

别忘记，在实际的项目中，很可能遇到缺失数据，于是得到的数据中就会有 NaN。

```
In [9]: city_gdp.loc['HongLouShi'] = np.nan    #杜撰的城市，若真的有，纯属巧合
        city_gdp.mean()
Out[9]: GDP            19319.79875
        Population      1828.25125
        dtype: float64
```

与 In[5]的计算对照，居然一样。为何？欲知原因，要看文档。

```
In [10]: city_gdp.mean?
Signature:  city_gdp.mean(axis=None,  skipna=None,  level=None,  numeric_only=None,
**kwargs)
……    #省略部分文档内容
skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA
……    #省略部分文档内容
```

这里有一个参数 skipna，默认是 True，即在 In[9]默认情况下将缺失的项略去了，不计入统计范畴。Pandas 的确为用户着想。

此外，还有一个 describe()方法，能够给出此对象的基本统计项目。

```
In [11]: city_gdp.describe()
Out[11]:
                          GDP    Population
         count       8.000000      8.000000
```

```
mean     19319.798750  1828.251250
std       4908.677545   645.654074
min      12170.200000  1137.870000
25%      17037.842500  1368.777500
50%      18688.995000  1576.940000
75%      20933.000000  2234.600000
max      27466.150000  3016.550000
```

Pandas 在统计运算方面的功能不仅有这些，还有更强悍的分组运算，这才是本节的重点。

2．分组运算

什么是分组？如果读者熟悉 SQL，对“分组”这个词语应该不陌生；如果读者不熟悉 SQL，则建议读者抽时间去学习，不过不影响对下述内容的理解。

如图 2-7-1 所示，对原来保存在“数据表”中的数据以“姓名”为键（也称为“关键词”）进行分组，并显示各组的结果，最终得到“期望结果”中的数据，这就是分组运算。

数据表

姓名	科目	分数
张三	物理	80
张三	数学	90
张三	英语	70
李四	物理	90
李四	数学	70
李四	英语	80

期望结果

姓名	物理	数学	英语
张三	80	90	70
李四	90	70	80

图 2-7-1　分组运算示例

在 SQL 中实现这个操作使用的是“group by”，那么在 Pandas 中怎样实现呢？

（1）分组对象

Pandas 中的“分组”，就其中文含义而言，可以有两种理解：①名词，已经按照某种方式划分的组；②动词，将某些对象按照某种方式进行划分。不管是“名词”还是“动词”，以 Python 的视角来看，它们都是“对象”——Python 中万物皆对象。

在具体研究分组对象之前，我们先了解一下在 Pandas 中对数据进行“分组”的操作过程，如图 2-7-2 所示。

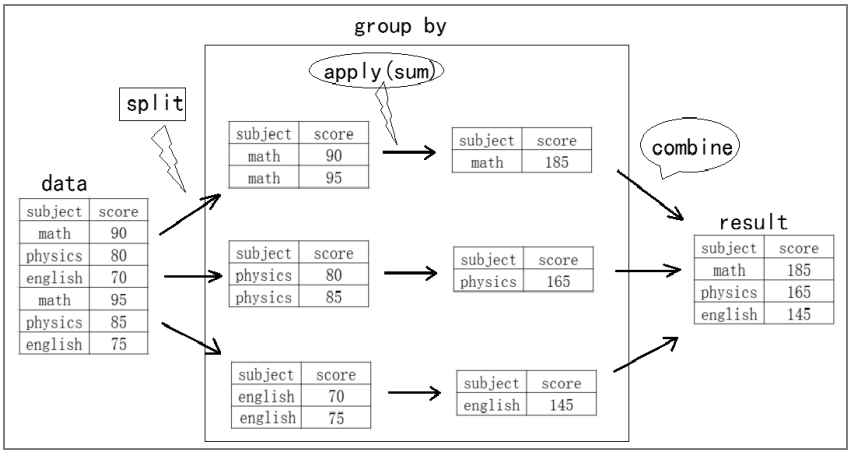


图 2-7-2　Pandas 中对数据进行“分组”的操作过程

假设有一个数据对象 data，它由“subject”和“score”两列标签（数据库表中的字段）及其数据组成，仔细观察“subject”列的数据，发现包含了三个不重复的字符串。如果以此列数据作为分组的键（按照图 2-7-2 中所示的数据），可以分为三组。完成分组后，再进行相关的运算。下面详细叙述 Pandas 中的分组（groupby）的运算过程。

① split（分割）：以指定的列（图 2-7-2 中指定为“subject”列）为键（关键词），将数据分割为若干组。这一步的执行结果得到了分组对象（图 2-7-2 中分别得到了三组对象）。

② apply（应用）：得到分组对象不是最终目的，一般情况下，要应用分组对象的某个方法对其数据进行计算。图 2-7-2 所示的示例中就是对分组对象中的数据进行求和。这一步执行的结果可能是一个标量，也可能是一个数组。

③ combine（合并）：得到各组的计算结果后，还要对各组的计算结果进行合并，以 DataFrame 或者 Series 对象的方式输出，这才是最终的计算结果。

在真实的运算过程中，上述三步是一气呵成的，请看下面的示例。

```
In [12]: df = pd.DataFrame({"subject":['math', 'physics', 'english', 'math', 'physics',
                                       'english'],
                            'score':[90, 80, 70, 95, 85, 75]})
         df
Out[12]:
            score    subject
         0     90       math
         1     80    physics
         2     70    english
         3     95       math
         4     85    physics
         5     75    english

In [13]: df.groupby("subject").sum()
Out[13]:
         subject  score
         english    145
         math       185
         physics    165
```

In[13]的操作是结果导向型的。所以，作为程序员，通常不需要关注中间的分组状态，将思维聚焦在最终得到的结果上即可。

当然，非要关心分组过程，也不是不可以。特别是在学习的过程中，对过程稍微了解一点比较好。

```
In [14]: sub_group = df.groupby("subject")
         sub_group
Out[14]: <pandas.core.groupby.DataFrameGroupBy object at 0x7fb4c5b701d0>
```

df.groupby("subject")指定以“subject”列的数据为键（分组的依据）对 df 进行分组，返回的不是 DataFrame，而是 DataFrameGroupBy，也可以把它理解为特殊的 DataFrame 对象。这个对象已经根据所指定的键对 df 进行了分组，但是还没有执行所指定的函数（例如 sum）操作，仅返回了这个对象。如果不进行后续的计算，此对象用处不大，或者说它的用武之地就是根据

后续所指定的函数进行计算。在英语中有一个很形象的词语来形容这类对象或者计算过程——lazy evaluation，翻译过来可以称之为“懒计算”或“延迟计算”。

In[14]执行某个集中量的函数完成 apply 的过程，然后自动合并各组计算结果，得到 Out[14]的结果。

刚才提到 sub_group 是一个对象，下面就从对象的角度进一步理解它。

```
In [15]: len(sub_group)
Out[15]: 3

In [16]: hasattr(sub_group, '__iter__')
Out[16]: True
```

In[15]中用函数 len()测量长度，得到的是分组对象中的元素个数。In[16]告诉我们，分组对象是可迭代的，既然如此，就可以对其实施循环操作了。

```
In [17]: for k, d in sub_group:
         print(k)
         print(d)

         english
            score  subject
         2     70  english
         5     75  english
         math
            score subject
         0     90    math
         3     95    math
         physics
            score  subject
         1     80  physics
         4     85  physics
```

虽然我们没有很直观地看到分组对象的内部，但是从 In[17]的输出结果中能推断出它是由两部分组成的（这就是黑箱方法）：一个是键，是组别的标识；另一个是相应组的数据。键和数据之间可以看作是一种对应关系。所以，也可以把分组对象转换为字典来观察。

```
In [18]: d = dict(list(sub_group))
         d['physics']
Out[18]:
             score  subject
         1      80  physics
         4      85  physics
```

可以直接输出转换后的字典（变量 d 引用的对象），虽然在视觉上有点乱，但是也能够正常输出。

除用 In[18]中的字典方式得到相应组的数据外，还可以使用分组对象的 get_group()方法。

```
In [19]: sub_group.get_group("physics")
Out[19]:
            score  subject
         1     80  physics
```

```
        4    85  physics
```

对于分组对象而言，我们常将它用于各种有关操作，比如前面演示的计算集中量等，当然不仅局限于此，后续将深入探究分组的依据，从而进行更复杂的运算。

（2）分组的键

为了理解分组，还是先完整呈现 groupby()方法的参数列表。

```
df.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False, **kwargs)
```

其中，参数 by 用来指明为当前数据对象进行分组的依据，即分组的键。能够作为键的数据类型包括字符串和其他可迭代对象、函数或其他表达对应关系的对象类型。

```
In [20]: df['teacher'] = ['Netwon', 'Netwon', 'Pascal', 'Netwon', 'Pascal', 'Pascal']
         df['rank'] = [4, 5, 2, 9, 7, 5]
         df
Out[20]:
            score  subject  teacher   rank
         0     90   math     Newton    4
         1     80   physics   Newton    5
         2     70   english   Pascal    2
         3     95   math      Newton   9
         4     85   physics   Pascal    7
         5     75   english   Pascal    5
In [21]: df.groupby('subject').mean()
Out[21]:
                   score  rank
         subject
         english    72.5   3.5
         math       92.5   6.5
         physics    82.5   6.0

In [22]: df.groupby(df['subject']).mean()
Out[22]:
                   score  rank
         subject
         english    72.5   3.5
         math       92.5   6.5
         physics    82.5   6.0

In [23]: type(df['subject'])
Out[23]: pandas.core.series.Series
```

In[21]是前面已经熟悉的操作，意思是以 df 的“subject”列的数据为键进行分组。In[22]中索性把“subject”列的数据（Series 对象）取出来，直接作为 groupby()方法的参数 by 的值。

除指定单一列的数据为键外，还可以指定多列的数据为键进行分组，下面就以“teacher”和“subject”两列数据为键进行分组。

```
In [24]: df.groupby(['teacher', 'subject']).mean()
Out[24]:
                           score  rank
```

```
        teacher   subject
        Newton    math      92.5   6.5
                  physics   80.0   5.0
        Pascal    english   72.5   3.5
                  physics   85.0   7.0
```

最终结果是一个具有多级索引的 DataFrame 对象。从多级索引中可以清晰地看出，已经依照我们所设定的分组键的顺序进行了分组。

此外，也可以指定 apply 环节的函数所计算的数据对象。

```
In [25]: df.groupby(['teacher', 'subject'])['score'].mean()
Out[25]: teacher  subject
         Newton    math      92.5
                   physics    80.0
         Pascal    english    72.5
                   physics    85.0
         Name: score, dtype: float64
```

用于分组的键不仅可以是数据本身，还可以是从外部找来的数据，只是不能随便找一个。请看下面的示例，特别注意其特点。

```
In [26]: data = pd.DataFrame(np.random.randn(6, 6), columns=["a", "b", "c", "d", "e", "f"],
                             index=["first", "second", "third", "forth", "fifth",
                                    "sixth"])
         data
Out[26]:
                         a         b         c         d         e         f
         first  -0.305505  0.851275 -0.123806 -0.783769 -0.872164  1.219267
         second  0.906511 -0.383093 -0.504328 -1.408830  0.623948 -0.126791
         third  -1.801244  0.359144 -1.370341  0.624486 -0.749871 -0.118133
         forth   0.749450  0.608357 -0.655950  0.454741 -0.808995  1.698974
         fifth  -1.237652  0.227126  0.178737  1.969729 -0.148190  0.090937
         sixth  -1.989540  0.948514  0.480491 -0.356188  1.360971  1.815460

In [27]: lst = ["one", "one", "one", "two", "two", "two"]    #作为分组的键
         data.groupby(lst, axis=1).mean()
Out[27]:
                        one       two
         first     0.140654 -0.145555
         second    0.006363 -0.303891
         third    -0.937480 -0.081173
         forth     0.233952  0.448240
         fifth    -0.277263  0.637492
         sixth    -0.186845  0.940081
```

In[27]中列表 lst 的长度和 data 的列标签个数一样（与 data 的行标签个数也一样），使用 lst 作为分组的键，并且声明 axis=1。从最终的结果可以看到，因为 lst 的长度与 data 的列标签个数相等，所以相当于 lst 的元素和 data.columns（列标签对象）的元素之间建立了一对一的对应关系，那么根据 lst 中的元素分组即可完成对 data 沿 1 轴方向分组。

接着上述分析思路，如果通过某种方式建立了 data.columns 的某种映射，也可以作为一种分组依据，那么字典就是建立映射的最佳数据结构。

```
In [28]: mapping = {'a':"one", 'b':"one", 'c':"one", 'd':"two", 'e':"two", 'f':"two"}
         data.groupby(mapping, axis=1).mean()
Out[28]:
                      one         two
         first   0.140654   -0.145555
         second  0.006363   -0.303891
         third  -0.937480   -0.081173
         forth   0.233952    0.448240
         fifth  -0.277263    0.637492
         sixth  -0.186845    0.940081
```

mapping 以字典的形式规定了一种映射，将此映射作为分组依据，得到了 Out[28]的结果。为了让读者对比 In[28]和 In[27]两个操作的结果，笔者都选择了在 axis=1 方向分组。读者可以自行建立一个映射，实现对 data 在 axis=0 方向的分组。

具有映射关系的对象除字典外，Series 也是一种——标签索引和数据之间也是一种映射关系。

```
In [29]: series_map = pd.Series(mapping)
         data.groupby(series_map, axis=1).mean()
Out[29]:
                      one         two
         first    0.140654   -0.145555
         second   0.006363   -0.303891
         third   -0.937480   -0.081173
         forth    0.233952    0.448240
         fifth   -0.277263    0.637492
         sixth   -0.186845    0.940081
```

综合以上用于分组的键，可以看出，它们都是可迭代对象——字符串、列表、Series 对象、字典映射等。除此之外，函数也可以作为分组的键——在 Python 中函数也是对象。

```
In [30]: def is_t(name):    #创建一个函数，判断对象中是否含有字母“t”
             if "t" in name:
                 return True
             else:
                 return False
         data.groupby(is_t).mean()
Out[30]:
                       a           b           c           d           e           f
         False  0.906511   -0.383093   -0.504328   -1.40883   0.623948   -0.126791
         True  -0.916898    0.598883   -0.298174    0.38180  -0.243650    0.941301
```

利用 is_t()函数对 data.index 中的标签索引进行判断，得到了一个迭代对象，然后这个迭代对象就作为分组的键，于是得到了 Out[30]的结果。当然，也可以使用 Python 内置函数。

```
In [31]: data.groupby(len).mean()
Out[31]:
                    a         b         c         d         e         f
         5   -0.916898   0.598883  -0.298174   0.38180  -0.243650   0.941301
         6    0.906511  -0.383093  -0.504328  -1.40883   0.623948  -0.126791
```

总之，最终能够作为键来对数据对象进行分组的，应该是一个可迭代对象。因此，下面所示的方式也就顺理成章了——在一个列表中包含多个分组键。

```
In [32]: index_lst = ['A', 'B', 'C', 'A', 'B', 'C']
         data.groupby([is_t, index_lst]).mean()
Out[32]:
                         a         b         c         d         e         f
         False B  0.906511 -0.383093 -0.504328 -1.408830  0.623948 -0.126791
         True  A  0.221972  0.729816 -0.389878 -0.164514 -0.840579  1.459121
               B -1.237652  0.227126  0.178737  1.969729 -0.148190  0.090937
               C -1.895392  0.653829 -0.444925  0.134149  0.305550  0.848663
```

不管表面上变量 by 所引用的对象是什么，归根结底能够用于分组的键都是我们已经熟悉的可迭代对象。当读者认识到此层面，对于分组键的选择就可以“不逾矩”了。

分组操作是分组运算的第一步，后面还有 apply（应用）呢，其花样也不少。

（3）分组对象的运算方法

用 dir()函数来查看分组对象，会看到分组对象的很多属性和方法，包括前面演示中出现的 sum()、mean()等。下面以一个分组对象的方法为例，实现对指定列的数据的运算。

```
In [33]: df.groupby('teacher')['score'].describe()
Out[33]:
                 count        mean       std   min    25%   50%    75%     max
         teacher
         Newton    3.0   88.333333  7.637626  80.0  85.0   90.0   92.5   95.0
         Pascal    3.0   76.666667  7.637626  70.0  72.5   75.0   80.0   85.0
```

在 In[33]中，规定了 describe()只计算 df['score']列的数据。从结果中的列标签可以看出，describe()得到了这么多项——常用的统计量。但是，如果不需要这么多统计项目，或者需要的统计项目这里还没有，应该怎么办？比如打算同时显示各组的标准差和平均值，显然不仅 describe()不能满足，分组对象的 std()和 mean()两个方法都不能直接满足。

Pandas 为分组对象提供了一些专有方法，如 aggregate()、apply()、filter()、transform()，专门用于解决分组运算中个性化的计算要求。注意，它们都是分组对象的方法，即它们都以分组对象为单位进行计算，此观点在后续的学习中还会以不同的方式进行重复。

① aggregate()

aggregate 有“聚合、聚集”之意，这里将它作为分组对象的方法名称。aggregate()函数的作用是可以将若干个用于统计运算的函数聚合到一起。

```
In [34]: df.groupby("teacher").aggregate(['mean', 'std', 'max'])
Out[34]:
                 score                      rank
                 mean       std      max   mean      std      max
         teacher
         Newton   88.333333   7.637626  95  6.000000  2.645751   9
         Pascal   76.666667   7.637626  85  4.666667  2.516611   7
```

在 In[34]中，使用分组对象的 aggregate()函数，将“平均数”、“标准差”和“最大值”三个计算函数聚合到了一起，让它们分别对两个分组对象中的数据进行相应的计算。比如，“score”列的最大值分别是“Newton”和“Pascal”两个分组中各自的最大值，不是整个 df 数据的最大

值。注意结果中的列标签是多级索引。

当然，如果 aggregate()的参数不是由多个函数对象组成的列表，而是一个函数对象，那么就与下面所示一样了。

```
In [35]: df.groupby('teacher').aggregate(np.median)
Out[35]:
                score   rank
         teacher
         Netwon     90    5
         Pascal     75    5

In [36]: df.groupby("teacher").median()
Out[36]:
                score  rank
        teacher
        Netwon      90    5
        Pascal      75    5
```

In[35]和 In[36]操作等效，当 aggregate()的参数仅仅是一个函数对象的时候，建议使用类似 In[36]的操作，更直接明了。

aggregate()还有一个别名 agg()。

```
In [37]: def distance(x):
         return x.max() - x.min()
         df.groupby('teacher')['score'].agg(distance)
Out[37]: teacher
         Netwon  15
         Pascal   15
         Name: score, dtype: int64
```

请读者观察上面的操作，有两个地方需要注意。

第一个是在 In[37]中，笔者使用 agg 替代了 aggregate，两者一样，只不过 agg 拼写简单罢了，然而在实践中笔者推荐使用 aggregate。

第二个是自定义的一个函数，此函数不是分组对象默认的方法，因此在 In[37]中再仿照 In[36]的样式写（df.groupby('teacher')['score'].distance()）肯定会报错，但又需要利用这个函数计算分组对象中的数据，所以就需要将函数对象 distance（这是函数对象，请不要写成 distance()）作为参数传入 agg()方法中（关于函数对象的具体内容，请参阅《跟老齐学 Python：轻松入门》的有关章节）。

分组对象的 aggregate()方法还能够对不同的列进行不同的运算，方法就是以字典形式建立列标签和函数对象的映射关系。

```
In [38]: df.groupby('teacher').aggregate({"score":np.mean, 'rank':np.median})
Out[38]:
                  score     rank
         teacher
         Netwon   88.333333    5
         Pascal   76.666667    5
```

```
In [39]: df.groupby('teacher').aggregate({"score":['max', 'min', np.mean],
                                          'rank':'std'})
Out[39]:
                  score                   rank
                  max  min    mean        std
        teacher
        Netwon     95   80   88.333333   2.645751
        Pascal     85   70   76.666667   2.516611
```

仔细观察上述操作中字典的“value”特点，更深刻领悟 aggregate()对各个分组进行的多函数运算（轴的方向是可以通过 axis 指定的，以上采用了默认值 axis=0）。

最后提示读者，在理解了上述阐述之后，请阅读 aggregate()的帮助文档。

② filter()

在理解 filter()方法之前，要明白函数名称的含义，因为编程中特别强调命名能够“望文生义”。filter 有“筛选”之意，具体如何“筛选”、“筛选”对象是谁，看看帮助文档或许能够理解一二。

```
Signature: df_group.filter(func, dropna=True, *args, **kwargs)
Docstring:
Return a copy of a DataFrame excluding elements from groups that do not satisfy the boolean
criterion specified by func.
```

有文档，必须配合示例，才能更明白。

```
In [40]: df     #继续使用 In[22]中所创建的 DataFrame 对象 df
Out[40]:
           score   subject  teacher  rank
        0     90   math      Netwon    4
        1     80   physics   Netwon    5
        2     70   english   Pascal    2
        3     95   math      Netwon    9
        4     85   physics   Pascal    7
        5     75   english   Pascal    5
```

df 所引用的 DataFrame 对象在前面示例中多次使用过了，这里再次展示全貌，目的是请读者将其与接下来的有关操作的结果进行比较。

```
In [41]: df.groupby('subject').mean()
Out[41]:
                   score  rank
        subject
        english     72.5    3.5
        math        92.5    6.5
        physics     82.5    6.0
```

In[41]的操作还属于复习，本着“温故知新”的谦虚态度，请仔细看看 Out[41]“rank”列的数据。下面逐渐进入“知新”阶段。

```
In [42]: def high_rank(x):
             return x['rank'].mean() > 5
```

这是一个函数，函数的返回值是布尔值，当然这个函数写法比较拙劣，它要求所输入的对

象必须有 x['rank']。

```
In [43]: df.groupby('subject').filter(high_rank)
Out[43]:
   score  subject  teacher  rank
0     90   math     Netwon    4
1     80   physics  Netwon    5
3     95   math     Netwon    9
4     85   physics  Pascal    7
```

将函数对象 high_rank 作为参数传入分组对象的方法 filter()内，意味着每个分组对象作为函数的参数 x 所引用的对象，比如“math”分组，在 high_rank()函数中参数 x 引用此分组对象，并计算该组中“rank”列数据的平均值，然后判断它是否大于 5。查看 Out[41]输出结果，发现此时函数返回值应该是 True。filter()以 True 为依据，将“math”组的数据以 DataFrame 方式返回。用同样的流程，依次检验“english”分组和“physics”分组。因为“english”分组的平均值小于 5，返回的是 False，所以 filter()不再返回该分组数据。最后，将返回的分组数据组合，得到了 Out[43]的结果。

filter()所筛选的对象是以分组对象为单位的，若符合条件，则该分组数据全部返回，不是对 Out[40]中的所有数据进行逐条筛选。

如果把前面那个拙劣的函数写得更 Pythonic 一些，可以用下面的方式表达。

```
In [44]: df.groupby('subject').filter(lambda x: x['rank'].mean() > 5)
Out[44]:
   score  subject  teacher  rank
0     90   math     Netwon    4
1     80   physics  Netwon    5
3     95   math     Netwon    9
4     85   physics  Pascal    7
```

最后提醒读者关注文档中的那句“Return a copy of a DataFrame”，即筛选后返回的结果是一个新的对象。

③ transform()

aggregate()指定针对分组对象运算的函数，得到的结果为分组对象各自运算结果的集合，此结果与原数据相比，不论是数据量还是行/列标签索引都发生了变化；filter()根据分组对象设置的条件对本分组对象中的数据进行筛选，最终将得到的数据合并返回，此返回结果一般比原数据中的记录少。相比而言，transform()的作用仅仅是“改变”了原数据的值，返回值的长度与原数据相同。特别要注意，返回值相对于原数据不是原地修改，而是生成一个新对象。

```
In [45]: df.groupby("subject").transform(lambda x: x - x.mean())
Out[45]:
   score  rank
0   -2.5  -2.5
1   -2.5  -1.0
2   -2.5  -1.5
3    2.5   2.5
4    2.5   1.0
5    2.5   1.5
```

在理解上述操作之前，先看一下 Out[40]和 Out[41]两个输出结果，一个是原始数据 df，另外一个是对 df 按照“subject”列分组后得到的各组的平均值。回顾前面这两个数据之后，再看 In[45]，分组对象的方法 transform()所操作的数据依然以分组对象为单位。这虽天经地义，但容易被忽视。传给此方法的函数对象（lambda x: x - x.mean()）针对每个分组对象进行计算。在上述例子中，用分组对象中的数据减去分组对象数据的平均值。以“math”分组为例，本分组数据的中“score”列的平均值是 92.5，“score”的数值有两个，即 df.iloc[0, 0]和 df.iloc[0, 3]，这两个值分别减去 92.5，就是 Out[45]中显示的[0, 0]和[0, 3]两个位置的数值。请读者在阅读上述内容的时候，用笔标记出相应的数据，便于理解其运算过程。

④ apply()

相对于前述的分组对象的各个方法，apply()比它们更灵活。还是先看示例，理解其灵活之所在。

```
In [46]: df_subject = df.groupby("subject")    #创建一个分组对象
         def rank_score(x):    #创建一个函数
             x['rank_score'] = x['score'] * x['rank'].std()
             return x
         df_subject.apply(rank_score)
Out[46]:
             score  subject    teacher  rank    rank_score
         0     90    math       Newton     4    318.198052
         1     80    physics    Newton     5    113.137085
         2     70    english    Pascal     2    148.492424
         3     95    math       Newton     9    335.875721
         4     85    physics    Pascal     7    120.208153
         5     75    english    Pascal     5    159.099026

In [47]: df_subject['rank'].std()    #分组对象中每组的“rank”列的标准差
Out[47]: subject
         english   2.121320
         math      3.535534
         physics   1.414214
         Name: rank, dtype: float64
```

在 In[46]的操作中，向分组对象的 apply()传入了函数对象，目的在于增加一列，并且所增加的列的数据按照 rank_score()函数的要求进行计算，最终得到 Out[46]的结果。依然强调，传入到 rank_score()中的参数是每个分组对象，所以计算中所需要的标准差就是 Out[47]所示的各组标准差。

这里仅以一个示例来讲解 apply()的应用方法，事实上它的灵活性还有其他层面的体现。本着“授人以渔”的原则，继续推荐读者在理解前述所有操作的基础上，阅读下述文档。

```
Signature: df_subject.apply(func, *args, **kwargs)
Docstring:
Apply function and combine results together in an intelligent way. The split-apply-combine
combination rules attempt to be as common sense based as possible. For example:

case 1:
group DataFrame
apply aggregation function (f(chunk) -> Series) yield DataFrame, with group axis having
```

```
group labels

case 2:
group DataFrame
apply transform function ((f(chunk) -> DataFrame with same indexes) yield DataFrame with
resulting chunks glued together

case 3:
group Series
apply function with f(chunk) -> DataFrame yield DataFrame with result of chunks glued
together
```

从某个角度来看，apply()的功能涵盖了前面的诸方法，那么是不是就能完全替代呢？不是的。以上各个函数之间还是有区别的。 区别在哪里呢？请读者特别关注参数列表（变量 df_subject 引用了一个分组对象）。

- df_subject.aggregate(arg, *args, **kwargs)
- df_subject.filter(func, dropna=True, *args, **kwargs)
- df_subject.transform(func, *args, **kwargs)
- df_subject.apply(func, *args, **kwargs)

“存在的都是合理的”，不管你是否认同，至少上面的方法都存在，虽然在某些情况下可以互相替代，但也有各自的特点。读者除通过本书前述内容理解上述这些方法的使用外，笔者还建议读者亲自通过本书反复强调的方法查看其文档。

在上述计算中，有一个细节不知道读者是否注意到，在所有对数值进行的计算中，数据为字符串的列自动不参与计算，尽管我们没有在任何位置声明，这就是 Pandas 的细致入微之处。

3．应用举例

1998 年，有一部电影红遍全国，名叫《泰坦尼克号》，虽然网上可以搜索到有关泰坦尼克号的相关数据的下载链接，但在这里，笔者使用来自 Seaborn 提供的数据集。

Seaborn 是一个专门用来进行数据可视化的库，官网网址是 https://seaborn.pydata.org/。借用该网站上的一句话来说明 Seaborn 的特点：Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphics.意思是 Seaborn 是一个基于 Matplotlib 的 Python 可视化库，它为绘制吸引人眼球的统计图提供了一个高级接口。第 3 章会专门讲解数据库可视化问题，这里仅使用 Seaborn 提供的泰坦尼克号的数据集。

首先安装 Seaborn，如果读者按照本书第 0 章所述，已经安装了 NumPy、SciPy、Pandas 和 Matplotlib，那么就满足了安装 Seaborn 的要求，否则请将上述库先安装好。确定已经有了上述依赖之后，使用 pip 命令来安装 Seaborn。

```
$ sudo pip3 install seaborn    #或者$ sudo pip install seaborn
```

安装成功后，切换到 Jupyter。

```
In [48]: import seaborn as sns
         titanic = sns.load_dataset('titanic')
         titanic.head()
Out[48]:
```

```
   survived  pclass     sex    age  sibsp  parch     fare  embarked  class
0         0       3    male   22.0      1      0   7.2500         S   Third
1         1       1  female   38.0      1      0  71.2833         C   First
2         1       3  female   26.0      0      0   7.9250         S   Third
3         1       1  female   35.0      1      0  53.1000         S   First
4         0       3    male   35.0      0      0   8.0500         S   Third

     who  adult_male  deck  embark_town  alive  alone
0    man        True   NaN  Southampton     no  False
1  woman       False     C  Cherbourg      yes  False
2  woman       False   NaN  Southampton    yes   True
3  woman       False     C  Southampton     yes  False
4    man        True   NaN  Southampton     no   True
```

下面运用分组对象，简单计算一下不同性别中因获救而幸存的平均人数。

```
In [49]: titanic.groupby('sex')['survived'].mean()
Out[49]: sex
         female    0.742038
         male      0.188908
         Name: survived, dtype: float64
```

果然，船上的男士们在生命攸关的时刻显示了绅士风度。

我们还可以用数据做进一步的观察。

```
In [50]: titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
Out[50]:
         class     First     Second   Third
         sex
         female   0.968085  0.921053  0.500000
         male     0.368852  0.157407  0.135447
```

In[50]也可以写成 titanic.groupby(['sex', 'class'])['survived'].mean().unstack()。

统计结果清楚地显示，头等舱就是好，不仅乘坐舒适，遇到事故幸存率也高。

从统计数据的角度看，对上述数据的统计，除使用分组运算外，还可以用透视表来完成。

```
In [51]: titanic.pivot_table('survived', index='sex', columns='class')
Out[51]:
         class    First     Second   Third
         sex
         female  0.968085  0.921053  0.500000
          male   0.368852  0.157407  0.135447
```

不管是用 groupby()，还是用 pivot_table()，都显示了一个冷冰冰的结果。

绅士风度不仅体现在对待女士上，在不同年龄层会怎样？下面按照未成年、中青年、老年三个层次进行统计，以 18、60、80 这三个年龄进行划分。

```
In [52]: ages = pd.cut(titanic['age'], [0, 18, 60, 80])
         titanic.pivot_table('survived', index=['sex', ages], columns='class')
Out[52]:
         class          First     Second   Third
```

```
sex    age
female  (0, 18]   0.909091  1.000000  0.511628
        (18, 60]  0.972222  0.900000  0.413793
        (60, 80]  1.000000       NaN  1.000000
  male  (0, 18]   0.800000  0.600000  0.215686
        (18, 60]  0.416667  0.061728  0.136364
        (60, 80]  0.083333  0.333333  0.000000
```

请读者认真浏览上面的数据，可能五味杂陈，可能有各种解读。

下面看一下 In[52]中所使用的一个新的 Pandas 方法 cut()吧。

```
Signature: pd.cut(x, bins, right=True, labels=None, retbins=False, precision=3,
include_lowest=False)
Docstring:
Return indices of half-open bins to which each value of `x` belongs.
```

pd.cut()方法返回的是 category 类型的对象，翻译为中文可以理解为“类别”类型的对象。下面通过示例来讲解。

```
In [53]: x = pd.Series([20, 25, 30, 35, 40])
         bins = [10, 30, 40]
         pd.cut(x, bins)
Out[53]: 0    (10, 30]
         1    (10, 30]
         2    (10, 30]
         3    (30, 40]
         4    (30, 40]
         dtype: category
         Categories (2, interval[int64]): [(10, 30] < (30, 40]]
```

变量 bins 引用的列表对象是做“箱子”的原料，当把它传给 pd.cut()里的参数 bins 后，在此函数里面就形成了(10, 30]和(30, 40]两个半开半闭的“箱”。x 所引用的 Series 对象中有一系列的数值，通过 pd.cut()函数，根据“箱”的数据范围，把这一系列的数值分别“装”入相应的“箱”里面。返回值就是与 x 中的数值对应的“箱”组成的 category 类型的对象——已经“装箱”（划分了“类别”）的对象。

如果把上面的通俗说明改用学术词语表述，就是以 bins 的序列为分段依据（一个“箱”就是一段），将 x 所引用对象的数据离散化。

当然，读者还可以继续学习其他的参数，比如 right=True，这是默认值，所得结果见 Out[53]；若 right=False，则相反。

此外，还有一个与 pd.cut()类似的方法 pd.qcut()，它能够根据数据的分位数进行离散化，具体使用方法请读者自行查看文档。

在理解了 pd.cut()方法之后，再来看 In[52]的操作，注意参数 index=['sex', ages]，这里指定的索引是以 Series(['female', 'male'])和 ages 生成的多级索引。

这里以泰坦尼克号的数据为例，运用已学知识，做了简要的数据分析。其实还可以做更多层面的分析和展示，请读者继续学习，后面会更精彩。

2.8 矢量化字符串

Python 的字符串操作功能已经很“强悍”了，而 Pandas 为字符串的操作提供了增强版的“强悍”。怎见得？看下面的示例。

```
In [1]: import numpy as np
        import pandas as pd
In [2]: name = "newton"
        name.capitalize()
Out[2]: 'Newton'
```

In[2]中变量 name 引用了一个普通的字符串，接着通过字符串的方法 capitalize()将字符串首字母变为大写——这在 Python 中是非常简单的操作。

```
In [3]: names = ['newton', 'hertz', 'curie']
        [name.capitalize() for name in names]
Out[3]: ['Newton', 'Hertz', 'Curie']
```

In[3]中变量 names 引用的是几个字符串组成的列表。如果让这个列表中的每个字符串的首字母都变成大写，用 In[3]中的列表解析式可以轻松完成——Python 对字符串的处理，至此还依然“强悍”。

```
In [4]: names2 = ['newton', 'hertz', None, 'curie']
```

In[4]中的 names2 虽然与前面的 names 不同，但也是一个正常的数据，不要认为多了一个 None 就不正常了，因为我们的研究对象是数据，所以数据中有缺失是再正常不过的事情。而就是这个所谓的正常，导致了不“正常”。

```
In [5]: [name.capitalize() for name in names2]
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-506-beadfb3bf714> in <module>()
----> 1 [name.capitalize() for name in names2]

<ipython-input-506-beadfb3bf714> in <listcomp>(.0)
----> 1 [name.capitalize() for name in names2]

AttributeError: 'NoneType' object has no attribute 'capitalize'
```

“强悍”的列表解析报错了。

再从我们熟悉的角度看，如果要处理的列表对象不是由字符串组成的，而是由数字组成的，会怎样？

```
In [6]: n = [1, 2, None, 4]
        ns = pd.Series(n)    #转换为 NumPy 的数组亦然
        ns + 5
Out[6]: 0    6.0
        1    7.0
        2    NaN
        3    9.0
        dtype: float64
```

以上是我们已经熟悉的做法，In[6]中的 pd.Series(n)操作，通常被称为将对象矢量化。矢量化之后，原对象中的 None 就变成了 NaN（一个特殊的浮点数），然后就可以进行有关运算了。

那么，In[4]中 names2 所引用的那个列表，能不能将其“矢量化”？当然可以，因为 Series 对象的元素不是非要求为数字。

```
In [7]: snames = pd.Series(names2)
        snames
Out[7]: 0    newton
        1      hertz
        2      None
        3      curie
        dtype: object
```

变量 snames 所引用的对象就不再是列表了，而是一个把列表矢量化为 Series 类型的对象。对于 Series 对象，有一个 str 属性，这个属性下面有一个方法，名称是 capitalize()，功能和前面的一样。

```
In [8]: snames.str.capitalize()
Out[8]: 0    Newton
        1      Hertz
        2      None
        3      Curie
        dtype: object
```

snames 和 ns 虽然都是 Series 对象，但有不同之处，比如：

```
In [9]: "str" in dir(ns),  "str" in dir(snames)
Out[9]: (False, True)
```

都是 Series 对象，但是“内涵”各异，导致拥有的属性有差别——snames 是矢量化字符串后的 Series 对象。

如果要查看 Pandas 为矢量化的字符串提供了哪些方法，可以使用“snames.str. + <Tab>”方式。

仔细观察，并与 dir(str)对比。理性的程序员，是不甘于用眼睛盯着看的。

```
In [10]: a = dir(str)    #字符串对象的所有属性和方法的名称列表
         b = dir(snames.str)    #矢量化字符串的所有属性和方法的名称列表
         da = [i for i in a if '_' not in i]    #去掉特殊属性和方法
         db = [i for i in b if '_' not in i]
         share = np.intersect1d(da, db)    #两者共有的方法名称
         share
Out[10]: array(['capitalize', 'center', 'count', 'encode', 'endswith', 'find', 'index',
                'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'islower', 'isnumeric',
                'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
                'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
                'rsplit', 'rstrip', 'split', 'startswith', 'strip', 'swapcase', 'title',
                'translate', 'upper', 'zfill'],
         dtype='<U12')
```

通过以上结果，可以说，矢量化字符串对象与 Python 中内置的字符串对象，它们所拥有的方法名称绝大多数是相同的，在进行字符串矢量化操作的时候，可以应用以往的经验。

要加深理解，就要重复多次。所以，请读者的思维和视线跟随下面的代码而行。

```
In [11]: phy = ['issac newton', 'albert Einstein', 'Max Karl Planck', 'Wilhelm Röntgen']
```

```
         phy = pd.Series(phy)
         phy.str.lower()
Out[11]: 0    issac newton
         1    albert einstein
         2    max karl planck
         3    wilhelm röntgen
         dtype: object

In [12]: phy.str.len()
Out[12]: 0    12
         1    15
         2    15
         3    15
         dtype: int64

In [13]: phy.str.split()
Out[13]: 0     [issac, newton]
         1     [albert, Einstein]
         2     [Max, Karl, Planck]
         3     [Wilhelm, Röntgen]
         dtype: object
```

In[11]中得到了一个矢量化字符串的 Series 对象 phy，后面会继续使用它。

```
In [14]: phy_mail = pd.Series(['new@itdiffer.com', 'ein@itdiffer.com',
                               'pla@itdiffer.com', 'ron@itdiffer.com'], index=phy)
         phy_mail['laoqi'] = np.NaN
         phy_mail
Out[14]: issac newton        new@itdiffer.com
         albert Einstein     ein@itdiffer.com
         Max Karl Planck     pla@itdiffer.com
         Wilhelm Röntgen     ron@itdiffer.com
         laoqi                     NaN
         dtype: object
```

对字符串的操作，正则表达式具有不可替代的作用（如果对正则表达式不熟悉，请搜索相关资料学习），对矢量化字符串也不例外。

```
In [15]: email_str = '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,4}$'
         phy_mail.str.match(email_str)
Out[15]: issac newton         True
         albert Einstein      True
         Max Karl Planck      True
         Wilhelm Röntgen      True
         laoqi                NaN
         dtype: object
```

In[15]中的字符串是电子邮件的正则表达式，将它用于 phy_mail.str.match()方法，如果与此正则表达式匹配，则返回 Ture——批量检验邮箱是否符合某种模式要求。

在对待矢量化字符串上，读者可以大胆测试字符串中的方法是否仍然适用，比如索引和切片。

```
In [16]: phy     # In[11]中所得
```

```
Out[16]: 0       issac newton
         1       albert Einstein
         2       Max Karl Planck
         3       Wilhelm Röntgen
         dtype: object

In [17]: phy.str[2]
Out[17]: 0   s
         1   b
         2   x
         3   l
         dtype: object

In [18]: phy.str.get(2)
Out[18]: 0   s
         1   b
         2   x
         3   l
         dtype: object

In [19]: phy.str[:5]
Out[19]: 0   issac
         1   alber
         2   Max K
         3   Wilhe
         dtype: object

In [20]: phy.str.slice(0, 5)
Out[20]: 0   issac
         1   alber
         2   Max K
         3   Wilhe
         dtype: object
```

从以上示例中可以看出，Python 字符串中的索引和切片方法，对于矢量化字符串，还能继续使用，并且又提供了 get()和 slice()两个方法实现同样的功能。

目前，我们学习的内容都以 Series 对象为例，读者可以在此基础上试试 DataFrame 对象，方法与上述一样。

本节所介绍的矢量化字符串的诸多方法，前文已经教给读者应该如何查看了。请读者务必认真查看，阅读其名称，如有必要进一步阅读其文档。

最后要说明的就是字符串矢量化会用在哪里呢？一般会在数据清洗的过程中使用。

2.9　与时间相关的操作

“一寸光阴一寸金，寸金难买寸光阴。”这是最形象、直观描写时间宝贵的箴言了。在数据分析中，时间也是一个非常重要的量，因此 Pandas 专门设置了与时间相关的对象。

Python 中有与时间有关的标准库，如 calendar、time、datetime 等，在《跟老齐学 Python：

轻松入门》一书中有专门阐述。

1．时间的物理量和对象

“时间”这个词有多种含义。例如，在“什么时间开始”和“持续多长时间”这两句话中，“时间”这个词所指的意思就不同，需要我们根据具体语境来体会其含义。这种模糊性在日常生活中一般不会有太大问题，甚至有时还是必需的，但是放到科学领域，就会麻烦不断了，科学讲究的就是准确。所以，科学中就有了很多专业术语，通过专业术语能够准确地表达概念和思想。

在物理学中，有三个物理量（“时刻”、“时间间隔”、“周期”）都是时间单位，或者说都是跟“时间”相关的物理量。下面就讲解 Pandas 中描述这三个物理量的专有对象类型。

（1）时刻

根据经典时空观，可以用一条有向直线表示时间，如图 2-9-1 所示。此数轴上的点表示“时刻”，两个点之间的距离表示“时间间隔”。

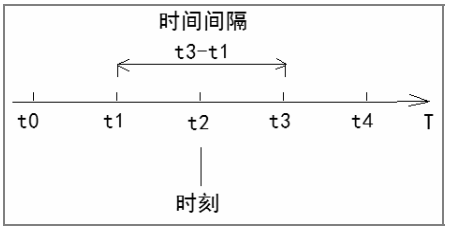


图 2-9-1　时刻与时间间隔

图 2-9-1 中数轴所表示的时间单位，可以是年、月、日、时、分、秒等。对于“时刻”而言，比如 t1=2017 年或者 t1=10 月等，都可以。也就是说，“时刻”是一个表示瞬时的量，而“瞬时”是一个极限的概念，它可以有不同的趋近。

在 Python 的 datetime 标准库中，也为“时刻”提供了专有的对象类型。

```
In [1]: import datetime
        now = datetime.datetime.now()    #当前“时刻”对象
        now
Out[1]: datetime.datetime(2017, 10, 11, 11, 8, 46, 636089)

In [2]: now.day, now.month, now.year
Out[2]: (11, 10, 2017)

In [3]: datetime.datetime.today()
Out[3]: datetime.datetime(2017, 10, 11, 11, 10, 29, 886083)
```

Pandas 中定义了一个名为 Timestamp 类型的对象来描述“时刻”这个物理量，并且 Timestamp 类继承了 datetime。

```
In [4]: import numpy as np
        import pandas as pd
        now = pd.Timestamp.now()    #Pandas 中的当前“时刻”对象
        now
Out[4]: Timestamp('2017-10-11 11:16:40.290771')
```

```
In [5]: now.year, now.month, now.day
Out[5]: (2017, 10, 11)
```

仅仅为了重复 datatime 中的功能而继承是没有价值的，也不是 Pandas 的最终目标。没有比较，看不到差别，所以先通过一段 Python 代码，来看一个貌似“悖论”的结果。

```
In [6]: import pytz    #时区设置的第三方库，安装 sudo pip install pytz
        brussels_tz = pytz.timezone('Europe/Brussels')
        shanghai_tz = pytz.timezone('Asia/Shanghai')
        now_brussels = datetime.datetime.now(tz=brussels_tz)
        now_shanghai = now_brussels.replace(tzinfo=shanghai_tz)
        now_shanghai == now_brussels
Out[6]: False
```

In[6]中的 now_brussels 为丹麦的布鲁塞尔时区的当前“时刻”数值，后续代码的目的是将 now_brussels 的数值转换为上海时区的数值。

根据常识理解，可以认为 In[6]中的 now_brussels 和 now_shanghai 所代表的是“时钟时间”，即在同一时刻看到的两个不同时区的当地时间钟表上的时刻，所以返回 False 是可以理解的。

根据经典时空观，这两个数据虽然是不同时区，但表示的是同一时刻；并且，尽管是 Python 中的两个对象，但 now_shanghai 是由 now_brussels 转换而来的，数值可以不同，但本质所指应该是同一个对象。换一个不是很恰当但容易理解的情形，有一根棍子长 1m，换算成厘米是 100cm，1 和 100 不等，但如果考虑这个具体情景，1m 和 100 厘米就相等了。

所以，In[6]貌似存在“悖论”。看 Pandas 的解决方法。

```
In [7]: now_shanghai = pd.Timestamp.now(tz="Asia/Shanghai")
        now_shanghai    #上海所在时区当前“时刻”的对象
Out[7]: Timestamp('2017-10-11 17:51:53.051739+0800', tz='Asia/Shanghai')

In [8]: now_brussels = now_shanghai.tz_convert("Europe/Brussels")
        now_brussels    #与 now_shanghai 同一时刻
Out[8]: Timestamp('2017-10-11 11:51:53.051739+0200', tz='Europe/Brussels')

In [9]: now_shanghai == now_brussels
Out[9]: True
```

now_shanghai 所引用的 Timestamp 对象描述的是上海时区的当前“时刻”，与此同时 Brussels 时区的 Timestamp 对象是 now_brussels，两个数值不同的对象描述的是同一个“时刻”。于是，In[9]中的比较结果是 True，反映的就是这个物理学本质。

注意观察，实现这种符合经典时空观下时刻转换的函数是 In[8]中的 tz_convert()。

除通过 Timestamp 类的 now()方法可以获得表示当前时刻的对象外，还可以通过实例化 Timestamp 类得到指定“时刻”的 Timestamp 对象，下面列出几种常用的实例化方式。

```
In [10]: pd.Timestamp(datetime.datetime(1997, 7, 1))
Out[10]: Timestamp('1997-07-01 00:00:00')

In [11]: pd.Timestamp("1997-7-1")
Out[11]: Timestamp('1997-07-01 00:00:00')
```

```
In [12]: pd.Timestamp(1997, 7, 1)
Out[12]: Timestamp('1997-07-01 00:00:00')
```

现代天体物理学已经告诉我们，这个宇宙有始有终，但是开始或者结束的时刻，我们是无法用现在习惯的几点几分来描述的。Pandas 则不然，它是人造的，人造的就有很多局限。

```
In [13]: pd.Timestamp.min
Out[13]: Timestamp('1677-09-21 00:12:43.145225')

In [14]: pd.Timestamp.max
Out[14]: Timestamp('2262-04-11 23:47:16.854775807')
```

目前来看，这个局限不会影响你的绝大多数应用。

以上都是不带有时区信息的 Timestamp 对象，如果带有时区信息，则需要通过参数 tz 进行声明。

```
In [15]: start = pd.Timestamp("1997-7-1", tz='Asia/Shanghai')
```

现在我们已经明确，“时刻”这个物理量在 Pandas 中使用 Timestamp 类型的对象表示。在这个基础上，我们可以进一步研究“时间间隔”——两个时刻的差就是一个时间间隔，俗称“时间段”。

（2）时间间隔

根据前述物理学中的定义，可以通过以下方式得到两个“时刻”之间的“时间间隔”。

```
In [16]: delta = now_shanghai - start    #这两个时刻都含有时区信息
         delta
Out[16]: Timedelta('7407 days 17:51:53.051739')
```

这里所看到的 Timedelta 类型对象，就是 Pandas 为描述“时间间隔”这个物理量而定义的一个对象类型。

在物理学中，如果一段时间间隔是Δt，则这个“时间间隔”可以和某个“时刻”做加法或者减法运算，即表示该“时刻”向前或者向后推移Δt 时间间隔，结果得到另外一个时刻。

```
In [17]: future = now_shanghai + delta
         future
Out[17]: Timestamp('2038-01-22 11:43:46.103478+0800', tz='Asia/Shanghai')
```

Timedelta 是 Pandas 中的一个类，于是就能实例化 Timedelta 类得到一个表示某“时间间隔”的对象——与 Timestamp 类似。

```
In [18]: pd.Timedelta(days=3, hours=4, minutes=5, seconds=6)
Out[18]: Timedelta('3 days 04:05:06')

In [19]: pd.Timedelta(seconds=123456)
Out[19]: Timedelta('1 days 10:17:36')
```

（3）周期

有的时间间隔有一定的特殊性，比如从 0 点到 24 点，每天都有这样一个时间间隔，周而复始。这种有规律的、不断重复的时间间隔，我们称之为周期。Pandas 中单独定义了一个对象来描述“周期”。

```
In [20]: now_week = pd.Period.now(freq="W")
         now_week
Out[20]: Period('2017-10-09/2017-10-15', 'W-SUN')
```

now_week 所引用的对象就是描述“周期”的 Period 对象，即 pd.Period.now(freq="W")创建了当前时刻所在的这个星期的 Period 对象——周期对象。还是跟前面一样，可以通过实例化 Period 类创建描述任何“周期”的 Period 对象。因为周期要有一定的时间单位，比如秒、分钟、时、天等，在 Pandas 的 Period 类中有一个重要的参数 freq（freq 的含义就是 frequency），通过它来确定该对象的周期频率，即以什么为单位。下面列出 Pandas 中认定的周期频率/单位符号。

```
In [21]: from pandas.tseries import frequencies
         frequencies._period_code_map.keys()
Out[21]: dict_keys(['S', 'A-JAN', 'M', 'W-THU', 'Q-JUN', 'A-DEC', 'A-JUL', 'W-MON',
                     'A-FEB', 'Q-OCT', 'Q-JAN', 'A-SEP', 'Q-DEC', 'W', 'B', 'T', 'U',
                     'Q-NOV', 'A-NOV', 'A-AUG', 'A-MAR', 'A-APR', 'A-JUN', 'Q', 'A-MAY',
                     'C', 'A-OCT', 'L', 'Q-APR', 'Q-MAY', 'W-SUN', 'D', 'A', 'W-WED', 'H',
                     'W-FRI', 'Q-SEP', 'Q-MAR', 'Q-AUG', 'Q-JUL', 'W-SAT', 'W-TUE', 'N',
                     'Q-FEB'])
```

观察 Out[20]的输出结果，包含了 now_week 这一周的起始和结束时刻，这两个“时刻”应该分别用 Timestamp 类型的对象描述，可以通过 now_week 的两个属性得到开始时刻和结束时刻的 Timestamp 对象。

```
In [22]: now_week.start_time, now_week.end_time
Out[22]: (Timestamp('2017-10-12 00:00:00'), Timestamp('2017-10-12 23:59:59.999999999'))
```

既然 Period 是 Pandas 中的类，也就能够把它实例化——请联想 Timestamp 和 Timedelta 类。

```
In [23]: pd.Period("1997-07")
Out[23]: Period('1997-07', 'M')

In [24]: pd.Period("1997-07", freq="D")
Out[24]: Period('1997-07-01', 'D')
```

可以通过 Timestamp 对象，借助 freq 所规定的频率创建 Period 对象。用物理学术语说明：假设某时刻 t1，若以“星期”为单位，则得到 t1 时刻所在的那个周（星期）。

```
In [25]: t1 = pd.Timestamp("1997-07-01")
         t1.to_period("W")
Out[25]: Period('1997-06-30/1997-07-06', 'W-SUN')
```

至此，物理学中的“时刻”、“时间间隔”和“周期”三个物理量分别被 Pandas 中专有的三个数据对象 Timestamp、Timedelta 和 Peroid 描述了，跟其他的 Python 对象一样，它们既是类，又是对象类型，并且实例有相应的属性。

```
In [26]: now = pd.Timestamp.now()    #当前时刻
         now.year, now.month, now.day    #当前时刻的年、月、日
Out[26]: (2017, 10, 12)

In [27]: now.dayofyear    #now.day 是一年中的第几天
Out[27]: 285

In [28]: now.dayofweek    #是本周的第几天，周一是第 0 天
```

```
Out[28]: 3

In [29]: now.hour    #此时刻是今天的第多少小时
Out[29]: 19
```

以上仅仅是简单举例，更多的属性和方法请读者查看文档。

Pandas 用三个对象来描述三个物理量，而在 Python 中对时间的操作，何止这些！

2．比较相关模块

在 Python 中，处理时间的方式比较多，为了便于后续的学习，这里有必要进行适当的梳理。当然，此处的梳理相对比较简单，主要目的是把 Python 不同模块所提供的操作放到一起，比较一下，才能看出它们的异同。

```
In [30]: datetime.datetime(year=1997, month=7, day=1)
Out[30]: datetime.datetime(1997, 7, 1, 0, 0)
```

datetime 是 Python 中的标准库，它提供了最基本并且常用的相关函数，读者可以参考《跟老齐学 Python：轻松入门》的相关章节。

此外，还有一个名为 dateutil 的标准库，它以 datetime 为基础，提供了一些强悍的扩展功能，比如解析字符串中的时间。

```
In [31]: from dateutil import parser
         date = parser.parse("1st of July, 1997")
         date
Out[31]: datetime.datetime(1997, 7, 1, 0, 0)

In [32]: parser.parse("1997.07.01")
Out[32]: datetime.datetime(1997, 7, 1, 0, 0)
```

dateutil.parser 号称能够解析所有的时间和日期字符串，然而当笔者输入 parser.parse("1997 年 7 月 1 日")时，它报错了（ValueError: Unknown string format）。不过，也要认同，它的确是有强悍的解析能力，如果读者有深入了解的想法，可以参考官方网站（http://dateutil.readthedocs.io/en/stable/）。

除能够把字符串解析为 datetime 外，还能反过来，比如下面的示例。

```
In [33]: date.strftime('%A')
Out[33]: 'Tuesday'
```

以上模块没有涉及到时区问题，前面已经使用第三方库 pytz 解决了时区问题，此处不再赘述。

虽然以上这些模块在处理日期和时间的操作中表现不俗，但毕竟都是针对单个数据操作的。在数据分析中，我们遇到的多是有着某种周期或者其他规律的时间序列。作为数据分析基础模块的 NumPy，也提供了一种针对日期/时间的专有数据类型 datetime64（第 1 章没有介绍它，这里补充）。

```
In [34]: date_np = np.array('1997-07-01', dtype=np.datetime64)    #注意字符串日期格式
         date_np
Out[34]: array(datetime.date(1997, 7, 1), dtype='datetime64[D]')
```

datetime64 类型的对象是以 64 位的精度存储的，像 date_np 那样。如果以“日”为单位（dtype='datetime64[D]'），则最大时间跨度就是 2^{64} 日。对于时间/日期的单位，如果不明确指定，就认为是所输入时间的最小单位，否则就以指定的单位为准。

```
In [35]: m = np.datetime64("1997-07")    #默认单位为“月”
         m, m.dtype
Out[35]: (numpy.datetime64('1997-07'), dtype('<M8[M]'))

In [36]: n = np.datetime64("1997-07-01 19:19:00")    #默认单位为“秒”
         n, n.dtype
Out[36]: (numpy.datetime64('1997-07-01T19:19:00'), dtype('<M8[s]'))

In [37]: n = np.datetime64("1997-07-01 19:19:00", 'D')    #指定单位为“日”
         n, n.dtype
Out[37]: (numpy.datetime64('1997-07-01'), dtype('<M8[D]'))

In [38]: n = np.datetime64("1997-07-01 19:19:00", 'ns')    #指定单位为“纳秒”
         n, n.dtype
Out[38]: (numpy.datetime64('1997-07-01T19:19:00.000000000'), dtype('<M8[ns]'))
```

NumPy 中以 datetime64 为数据类型所创建的数据，在处理一系列的日期/时间上就表现出优势了。

```
In [39]: date_np + np.arange(20)
Out[39]: array(['1997-07-01', '1997-07-02', '1997-07-03', '1997-07-04',
                '1997-07-05', '1997-07-06', '1997-07-07', '1997-07-08',
                '1997-07-09', '1997-07-10', '1997-07-11', '1997-07-12',
                '1997-07-13', '1997-07-14', '1997-07-15', '1997-07-16',
                '1997-07-17', '1997-07-18', '1997-07-19', '1997-07-20'],
                dtype='datetime64[D]')
```

如此，我们就能轻易地以矢量形式表示日期/时间了。

虽然以 NumPy 数组的 datetime64 类型数据实现了上述时间序列，但是我们也要清醒地认识到，相对于使用 datetime 和 dateutil，在使用的便捷性上还有差距——这个问题必须解决，于是 Pandas 的方法就出现了。

```
In [40]: date_pd = pd.to_datetime("1st of July, 1997")
         date_pd
Out[40]: Timestamp('1997-07-01 00:00:00')

In [41]: date_pd.strftime('%A')
Out[41]: 'Tuesday'

In [42]: date_pd + pd.to_timedelta(np.arange(20), 'D')
Out[42]: DatetimeIndex(['1997-07-01', '1997-07-02', '1997-07-03', '1997-07-04',
                        '1997-07-05', '1997-07-06', '1997-07-07', '1997-07-08',
                        '1997-07-09', '1997-07-10', '1997-07-11', '1997-07-12',
                        '1997-07-13', '1997-07-14', '1997-07-15', '1997-07-16',
                        '1997-07-17', '1997-07-18', '1997-07-19', '1997-07-20'],
                        dtype='datetime64[ns]', freq=None)
```

Pandas 果然是“Dragon Warrior”，它的“神功”还会在索引上彰显。

3．时间索引

仔细观察 In[42]所得到的结果，其类型为 DatetimeIndex，这是以 Timestamp 对象为基础构建的新的数据类型——用于时间索引。

```
In [43]: datas = [pd.Timestamp('1997-7-1'), pd.Timestamp('1997-7-2'), pd.Timestamp
                 ('1997-7-3')]
         s = pd.Series([100, 105, 110], index=datas)
         s
Out[43]: 1997-07-01    100
         1997-07-02    105
         1997-07-03    110
         dtype: int64

In [44]: s.index
Out[44]: DatetimeIndex(['1997-07-01', '1997-07-02', '1997-07-03'],
                       dtype='datetime64[ns]', freq=None)
```

上述方式虽然也能得到 DatetimeIndex 对象，但显然不如使用 pd.to_datetime()更灵活。

```
In [45]: date_index = pd.to_datetime([datetime.datetime(1997,7,1), '3rd of July, 1997',
                                      '1997.7.5','1997-07-07','1997-Jul-8','19970709',
                                      '10/7/1997'])
         date_index
Out[45]: DatetimeIndex(['1997-07-01', '1997-07-03', '1997-07-05', '1997-07-07',
                        '1997-07-08', '1997-07-09', '1997-10-07'],
                        dtype='datetime64[ns]', freq=None)
```

在 pd.to_datetime()中，如果传入一个表示日期/时间的对象（可以是多种形式），则生成 Timestamp 对象；如果传入由表示“时刻”的数据组成的序列，则生成 DatetimeIndex 类型的索引对象。按照这个逻辑，是不是由表示“周期”的数据类型 Period 对象组成的序列，就是 PeriodIndex 类型的索引对象了呢？

```
In [46]: periods = [pd.Period('1997-07'), pd.Period('1997-08'), pd.Period('1997-09'),
                    pd.Period('1997-10')]
         ps = pd.Series(np.linspace(100, 200, 4), index = periods)
         ps
Out[46]: 1997-07    100.000000
         1997-08    133.333333
         1997-09    166.666667
         1997-10    200.000000
         Freq: M, dtype: float64

In [47]: ps.index
Out[47]: PeriodIndex(['1997-07', '1997-08', '1997-09', '1997-10'], dtype='period[M]',
freq='M')
```

猜对了，并且 Pandas 还提供了一个将 DatetimeIndex 索引对象转换为 PeriodIndex 索引对象的方法。或许是因为通过 pd.to_datetime()可以接收各类参数，致使创建 DatetimeIndex 对象更便捷吧！

```
In [48]: period_index = date_index.to_period("D")    #date_index 见 In[45]
         period_index
```

```
Out[48]: PeriodIndex(['1997-07-01', '1997-07-03', '1997-07-05', '1997-07-07',
                      '1997-07-08', '1997-07-09', '1997-10-07'],
                      dtype='period[D]', freq='D')

In [49]: period_index_m = date_index.to_period("M")
         period_index_m
Out[49]: PeriodIndex(['1997-07', '1997-07', '1997-07', '1997-07', '1997-07', '1997-07',
                      '1997-10'],
                      dtype='period[M]', freq='M')
```

通过 DatetimeIndex 对象的 to_period()方法，并且在参数中以 freq 声明“周期”的单位（频率），即可转换为 PeriodIndex 索引对象。

应该还有与 Timedelta 对应的索引对象，其名称可能是 TimedeltaIndex。这就不是猜测了，是不完全归纳得到的推论，不过要通过真实的操作才能检验是否正确。

```
In [50]: t = date_index[0]    #得到表示“时刻”的 Timestamp 对象
         t
Out[50]: Timestamp('1997-07-01 00:00:00')

In [51]: delta_index = date_index - t    #两个时刻差是时间间隔
         delta_index
Out[51]: TimedeltaIndex(['0 days', '2 days', '4 days', '6 days', '7 days', '8 days', '98
                         days'],
                         dtype='timedelta64[ns]', freq=None)
```

下面对上述操作做一个简单的总结。

- “时刻”，在 Pandas 中用 Timestamp 对象表征。用 Timestamp 对象可以构建 DatetimeIndex 索引对象。
- “周期”，在 Pandas 中用 Period 对象表征。用 Period 对象可以构建 PeriodIndex 索引对象，并且此对象也能够通过 DatetimeIndex 对象的 period()方法得到。
- “时间间隔”，在 Pandas 中用 Timedelta 对象表征。用 Timedelta 对象可以构建 TimedeltaIndex 索引对象。

有时候，作为索引的日期/时间是有规律的，类似使用 np.arange()生成连续的数值，可以使用 pd.date_range()、pd.period_range()和 pd.timedelta_range()生成三种索引对象，其中元素所表征的物理量是按照一定规律排列的。

```
In [52]: pd.date_range("1997-7", "1997-11")
Out[52]: DatetimeIndex(['1997-07-01', '1997-07-02', '1997-07-03', '1997-07-04',
                        '1997-07-05', '1997-07-06', '1997-07-07', '1997-07-08',
                        '1997-07-09', '1997-07-10',
                        ...
                        '1997-10-23', '1997-10-24', '1997-10-25', '1997-10-26',
                        '1997-10-27', '1997-10-28', '1997-10-29', '1997-10-30',
                        '1997-10-31', '1997-11-01'],
                        dtype='datetime64[ns]', length=124, freq='D')
```

本来想通过 pd.date_range("1997-7", "1997-11")得到以“月”为频率的 DatetimeIndex，结果却是 freq='D'，为何如此？请耐心看文档（下面仅列出此方法的参数列表）。

```
pd.date_range(start=None, end=None, periods=None, freq='D', tz=None, normalize=False,
name=None, closed=None, **kwargs)
```

在参数列表中我们找不到 step 这个参数（np.arange()中有），而是看到了 freq='D'，即默认生成的 DatetimeIndex 对象的频率是“日”。要得到频率为“月”的对象，需要在参数中声明。

```
In [53]: pd.date_range("1997-7", "1997-11", freq="M")
Out[53]: DatetimeIndex(['1997-07-31', '1997-08-31', '1997-09-30', '1997-10-31'],
                       dtype='datetime64[ns]', freq='M')
```

再看参数列表中的 periods，它的作用不是生成 PeriodIndex 对象，而是设置即将生成的 DatetimeIndex 对象中“时刻”数量。观察下面的示例。

```
In [54]: pd.date_range("2017-7-1", periods=8, freq="H")
Out[54]: DatetimeIndex(['2017-07-01 00:00:00', '2017-07-01 01:00:00',
                        '2017-07-01 02:00:00', '2017-07-01 03:00:00',
                        '2017-07-01 04:00:00', '2017-07-01 05:00:00',
                        '2017-07-01 06:00:00', '2017-07-01 07:00:00'],
                        dtype='datetime64[ns]', freq='H')
```

pd.period_range()和 pd.timedelta_range()的使用方法与 pd.date_range()的使用方法类似，读者可以在理解下面的示例之后，查看其文档，从而全面了解使用时的注意事项。

```
In [55]: pd.period_range("2017-7", periods=7, freq="M")
Out[55]: PeriodIndex(['2017-07', '2017-08', '2017-09', '2017-10', '2017-11', '2017-12',
                      '2018-01'],
                      dtype='period[M]', freq='M')

In [56]: pd.timedelta_range(0, periods=7)
Out[56]: TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days', '5 days', '6
                         days'],
                         dtype='timedelta64[ns]', freq='D')
```

freq 这个参数在很多函数中都会用到，它表示的是“周期”单位（频率），虽然在前面列出了它所有的值，但有的读者可能没有认真研究，或者只是扫一眼，连一句“这么多”的感叹都没有，就匆匆而过了。所以，推荐读者到前面“周期”相关内容那里再复习一番，借助搜索引擎，理解各值的含义（推荐阅读：http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases），然后在 Jupyter 中测试。

```
In [57]: pd.timedelta_range(0, periods=7, freq="3H30T")
Out[57]: TimedeltaIndex(['00:00:00', '03:30:00', '07:00:00', '10:30:00', '14:00:00',
                         '17:30:00', '21:00:00'],
                         dtype='timedelta64[ns]', freq='210T')

In [58]: pd.date_range("2017-9-30", periods=9, freq=pd.tseries.offsets.BDay())
Out[58]: DatetimeIndex(['2017-10-02', '2017-10-03', '2017-10-04', '2017-10-05',
                        '2017-10-06', '2017-10-09', '2017-10-10', '2017-10-11',
                        '2017-10-12'],
                        dtype='datetime64[ns]', freq='B')
```

在 In[58]中，使用了 pd.tseries.offsets.BDay()方法，其实前面我们为 freq 传入的符号所表示的值都可以看作是 pd.tseries.offsets 中的各个方法的“代号”。提醒读者，“B”和 Bday()表示的

频率是“工作日”，但是因为美国不过我国的国庆日，也没有把我国的中秋节作为法定假日，所以显示的结果并非是我们所期盼的。

4．重采样

“采样”（Sample）的意思是从大量对象中抽取一部分样品，这是一个统计学中的术语。我们在这里将要学习的是“resample()”，从这个方法的名称 resample 可以猜到它的基本功能应该是实现“重新采样”（简称“重采样”）。别瞎猜了，还是看看文档（注意：resample 是 Series、DataFrame 对象的方法，下面的示例中使用了 In[46]中所创建的 ps 对象）。

```
Signature: ps.resample(rule, how=None, axis=0, fill_method=None, closed=None, label=None,
convention='start', kind=None, loffset=None, limit=None, base=0, on=None, level=None)
Docstring:
Convenience method for frequency conversion and resampling of time series.  Object must
have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass
datetime-like values to the on or level keyword.
```

从文档中能够看出，可以进行重采样的对象必须是以 DatetimeIndex、PeriodIndex 或 TimedeltaIndex 为索引的。重采样做的主要事情就是变更这些索引的频率，例如下面的示例。

```
In [59]: dindex = pd.date_range("20/7/2017", periods=50, freq="D")
         sdata = pd.Series(np.random.randn(len(dindex)), index=dindex)
         sdata.index[0]
Out[59]: Timestamp('2017-07-20 00:00:00', freq='D')

In [60]: sdata.index[-1]
Out[60]: Timestamp('2017-09-07 00:00:00', freq='D')
```

上述操作中得到 sdata 这个 Series 对象的索引是 dindex，其时间范围是 2017-07-20 00:00:00—2017-09-07 00:00:00，并且频率是“日”（freq='D'）。可以将频率改为“月”，如下所示。

```
In [61]: sdata.resample('M').mean()
Out[61]: 2017-07-31    0.397003
         2017-08-31    -0.178797
         2017-09-30    -0.619291
         Freq: M, dtype: float64

In [62]: sdata.resample('M', kind='period').mean()
Out[62]: 2017-07    0.397003
         2017-08    -0.178797
         2017-09    -0.619291
         Freq: M, dtype: float64
```

原数据 sdata 的时间跨度中包括了 7、8、9 这三个月，在 In[61]的 resample()参数中确定以“月”为单位（将原来的频率'D'变更为'M'）输出数据，但是每个月都有那么多数据，输出哪一个呢？于是后面紧跟一个 mean()方法，意思是计算每个月数据的平均值。注意比较 Out[61]和 Out[62]的输出结果，其区别在于 kind 这个参数。表 2-9-1 列出了 resample()的部分参数及其说明，供读者参考。

表 2-9-1　resample()的部分参数及其说明

参　数	说　明
rule	转换目标的频率
how=None	用于计算集中量的函数对象，现已弃用。请使用 In[97]的方式
fill_method=None	重采样的插值方式：“ffill”或“bfill”
closed=None	{'right', 'left'}，哪一端闭合
label=None	{'right', 'left'}，设置哪个标签来标记
convention='start'	{'start', 'end', 's', 'e'}，将低频率转换到高频率所采用的约定
kind=None	{‘period’, ‘timestamp’}，聚合的类型
loffset=None	校正时间标签
level=None	对于多级索引，指定采样的索引层级

在 In[61]的操作中，相对原来的 sdata 而言，频率从“日”（D）变成了“月”（M），这种情况称为频率降低，又称为“降采样”。

```
In [63]: t_index = pd.date_range("2017-11-10", periods=20, freq="T")
         sdata2 = pd.Series(np.arange(20), index=t_index)
         sdata2.resample('7min').sum()
Out[63]: 2017-11-10 00:00:00    21
         2017-11-10 00:07:00    70
         2017-11-10 00:14:00    99
         Freq: 7T, dtype: int64

In [64]: sdata2.resample('7min', closed='right').sum()  #注意观察 closed='right'的效果
Out[64]: 2017-11-09 23:53:00     0
         2017-11-10 00:00:00     28
         2017-11-10 00:07:00     77
         2017-11-10 00:14:00     85
         Freq: 7T, dtype: int64
```

能够完成类似 resample()功能的实现频率转换的函数还有一个，名为 asfreq()。

```
In [65]: sdata2.asfreq('7min')
Out[65]: 2017-11-10 00:00:00     0
         2017-11-10 00:07:00     7
         2017-11-10 00:14:00    14
         Freq: 7T, dtype: int64
```

同样是“降采样”，asfreq()和 resample()有所不同，asfreq()返回了“7min”末那个时刻的值，请注意观察上述结果，并查看其帮助文档。

```
In [66]: sdata2.asfreq('S')
Out[66]: 2017-11-10 00:00:00     0.0
         2017-11-10 00:00:01     NaN
         2017-11-10 00:00:02     NaN
         …
         2017-11-10 00:18:58     NaN
         2017-11-10 00:18:59     NaN
         2017-11-10 00:19:00    19.0
         Freq: S, Length: 1141, dtype: float64
```

```
In [67]: sdata2.asfreq('S', method='bfill')
Out[67]: 2017-11-10 00:00:00    0
         2017-11-10 00:00:01    1
         2017-11-10 00:00:02    1
         …
         2017-11-10 00:18:58   19
         2017-11-10 00:18:59   19
         2017-11-10 00:19:00   19
         Freq: S, Length: 1141, dtype: int64

In [68]: sdata2.asfreq('S', method='ffill')
Out[68]: 2017-11-10 00:00:00    0
         2017-11-10 00:00:01    0
         2017-11-10 00:00:02    0
         …
         2017-11-10 00:18:58   18
         2017-11-10 00:18:59   18
         2017-11-10 00:19:00   19
         Freq: S, Length: 1141, dtype: int64
```

In[66]、In[67]和 In[68]都是“升采样”，对于空缺数据，默认是 NaN，或者按照 method 声明的方法填充，还可以使用另外一个参数 fill_value，直接指定填充数据。

```
In [69]: sdata2.asfreq('S', fill_value=999)
Out[69]: 2017-11-10 00:00:00     0
         2017-11-10 00:00:01   999
         2017-11-10 00:00:02   999
         …
         2017-11-10 00:18:58   999
         2017-11-10 00:18:59   999
         2017-11-10 00:19:00    19
         Freq: S, Length: 1141, dtype: int64
```

同样的“升采样”操作，读者也可以使用 resample()完成，此处不再重复，请自行测试。

通过上面的操作，读者是否理解“重采样”的含义了？其实，如果再复习一下分组运算，会发现 groupby()也是在“重采样”。

从索引是 DatetimeIndex、PeriodIndex 或 TimedeltaIndex 的数据中获得部分数据，直接使用下标也能实现一种“采样”。

```
In [70]: sdata['2017-09']    #sdata 见 In[57]中所创建的对象
Out[70]: 2017-09-01   -0.768886
         2017-09-02   -0.825489
         2017-09-03    0.661488
         2017-09-04   -0.019393
         2017-09-05   -0.458384
         2017-09-06   -1.950523
         2017-09-07   -0.973850
         Freq: D, dtype: float64
```

sdata 的索引是 DatetimeIndex 类型，频率是“日”（D），当下标表示“月”的时候，会返回

该月的所有数据。

其实，“采样”是一种比较综合的操作，以前学过的很多方法都能实现，不仅局限于此处介绍的这些。

2.10 简单的应用示例

虽然笔者用了不少的篇幅来讲解 Pandas，但也仅仅是九牛一毛，Pandas 中提供的工具非常多，而且它还在不断发展中。所以，不要奢望本书能够面面俱到，本书的职责也不是 Pandas“大全”。通过前面的学习，读者应该掌握了一些基本知识和技能，在此基础上，可以使用笔者反复提及的方法进行自我学习。

本节给读者展示的示例中，可能有新的方法出现。一方面笔者会做必要的解释，另一方面读者也要边阅读边使用帮助文档、Google 工具辅助理解。

1．准备工作

为了完成本节的综合应用，我们还需要安装一些模块，这也是 Python 在数据分析领域大行其道的原因，能够信手捏来的工具有很多。

首先安装 pandas-datareader，它是专门为 Pandas 提供经济数据的模块，其中包括来自 Yahoo、Google 等的财经数据，其官方网站为 https://pandas-datareader.readthedocs.io。安装方法可以参考下述操作，或者使用官网中提供的其他方法。

```
$ sudo pip3 install pandas-datareader
```

然后，安装支持绘图的 Matplotlib 库。

```
$ sudo pip3 install matplotlib     #检查在第 0 章是否已经安装
```

为了支持绘图结果的显示，还需要安装 tkinter 模块，它是专门用于 GUI 的第三方库。

```
$ sudo apt-get install python-tk
```

确定以上内容安装完毕，就要进行股票数据分析了。

进入到 Jupyter 交互环境后，把需要的模块引入。

```
In [1]: %matplotlib inline
        import numpy as np
        import pandas as pd
        import pandas_datareader
        import matplotlib.pyplot as plt
        import seaborn
        seaborn.set()
```

%matplotlib inline 旨在 Jupyter 交互环境中实现直接绘图，这种用法在后续数据可视化中常用。

首先要获得有关的股票数据，这里显示的是一种方式（注意：读者按照本书的方式获得数据，不一定能够成功，失败的原因很多，也不可描述，如果非要这些数据不可，可以到网上搜索有关财经数据的官方网站，一般都会提供数据接口或者数据文件下载）。

```
In [2]: aapl = pandas_datareader.data.DataReader("AAPL", 'yahoo')
        aapl.head()
```

```
Out[2]:
        Date  Open       High        Low      Close        Adj      Close     Volume
        2009-12-31  30.447144  30.478571  30.080000  30.104286  26.986492   88102700
        2010-01-04  30.490000  30.642857  30.340000  30.572857  27.406532  123432400
        2010-01-05  30.657143  30.798571  30.464285  30.625713  27.453915  150476200
        2010-01-06  30.625713  30.747143  30.107143  30.138571  27.017223  138040000
        2010-01-07  30.250000  30.285715  29.864286  30.082857  26.967278  119282800

In [3]: goog = pandas_datareader.DataReader("GOOG", 'google')
        goog.head()
Out[3]:
        Date  Open       High        Low       Close         Adj       Close   Volume
        2009-12-31  310.356445  310.679321  307.986847  307.986847  307.986847   2455400
        2010-01-04  311.449310  312.721039  310.103088  311.349976  311.349976   3937800
        2010-01-05  311.563568  311.891449  308.761810  309.978882  309.978882   6048500
        2010-01-06  310.907837  310.907837  301.220856  302.164703  302.164703   8009000
        2010-01-07  302.731018  303.029083  294.410156  295.130463  295.130463  12912000
```

得到了股票代码为 AAPL 和 GOOG 两个公司的部分股票数据，下面就对其进行处理。

2．处理股票数据

对于标题，感觉还是不用“分析”而用“处理”更好，以免误导读者，因为下面的确“分析”不出什么，更不能告诉读者如何购买股票——股市有风险，入市需谨慎。

```
In [4]: import datetime
        goog_close = goog.loc[datetime.date(2016, 1, 1):]['Close']
        goog_close.plot()
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7fcdeb2d0e48>
```

（输出结果如图 2-10-1 所示）

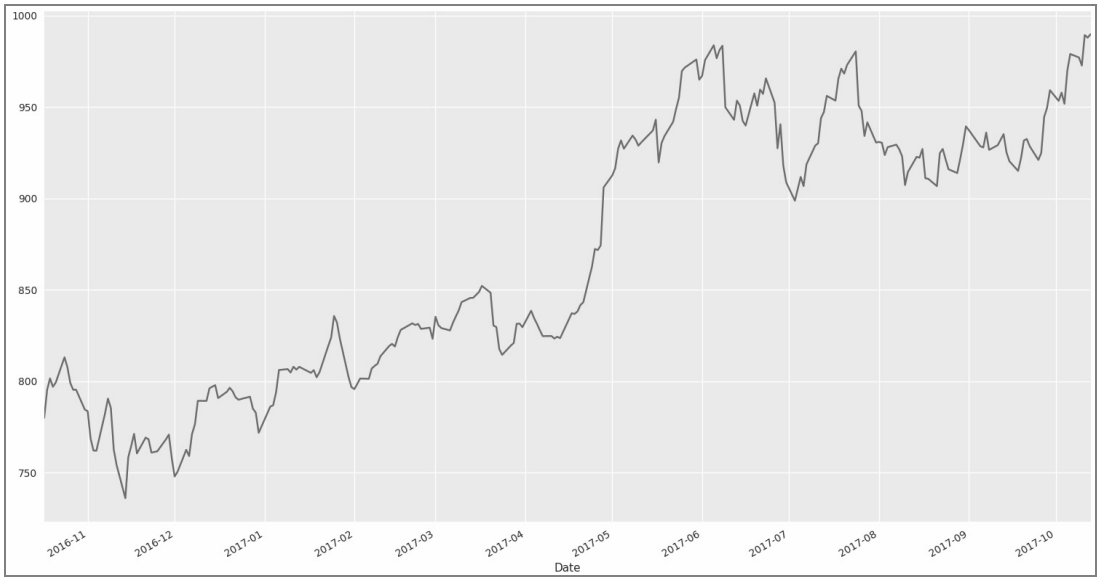


图 2-10-1　GOOG 股票收盘价变化

In[4]得到了 GOOG 股票的收盘价，然后进行绘图，我们可以很直观地看到自 2016 年 1 月 1 日以来交易日的收盘价变化曲线图。

注意，因为 In[2]的数据集随着时间在不断变化，所以读者依照本书进行代码调试，得到的图示或许与示例中的图示不同，这是因为数据集中又有了新的数据。比如 2018 年的数据在本书中就没有，当本书出版的时候，已经 2018 年了。

下面使用 resample()函数，计算每个月的平均收盘价，然后进行绘图。

```
In [5]: goog_close.plot(style='-')
        goog_close.resample('M').mean().plot(style=":")
        plt.legend(['closed', 'mean'], loc='upper left')
Out[5]: <matplotlib.legend.Legend at 0x7fcdea644a20>
```

（输出结果如图 2-10-2 所示）

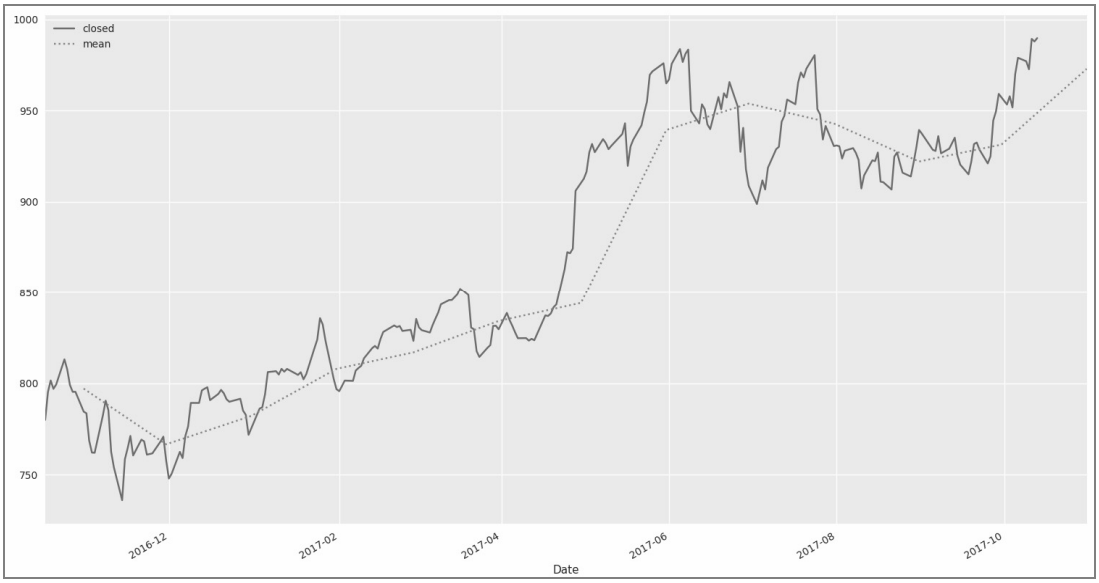


图 2-10-2　每月平均收盘价和每日收盘价

在股票数据分析中，“移动平均线”（Moving Average，简称 MA，也叫均线）是普遍应用的技术指标之一，它将某一周期内的收盘价之和除以该周期的时间间隔。应用 Series 和 DataFrame 对象的 rolling()方法可以计算 MA。

```
In [6]: rolling = goog_close.rolling(window=10, center=True)
        data = pd.DataFrame({'Closed':goog_close, 'rolling_mean': rolling.mean()})
        data.head(20)
Out[6]:
        Date            Closed   rolling_mean
        2016-01-04    741.840027   NaN
        2016-01-05    742.580017   NaN
        2016-01-06    743.619995   NaN
        2016-01-07    726.390015   NaN
        2016-01-08    714.469971   NaN
        2016-01-11    716.030029   722.073004
        2016-01-12    726.070007   718.067999
```

```
2016-01-13    700.559998   713.654999
2016-01-14    714.719971   709.952002
2016-01-15    694.450012   709.838001
2016-01-19    701.789978   709.558002
2016-01-20    698.450012   709.258997
2016-01-21    706.590027   706.650995
2016-01-22    725.250000   709.690997
2016-01-25    711.669983   712.514001
2016-01-26    713.039978   718.269000
2016-01-27    699.989990   724.555005
2016-01-28    730.960022   727.405005
2016-01-29    742.950012   727.547003
2016-02-01    752.000000   723.379004
```

In[6]中计算的是 MA10，即 10 天的短期移动平均线。观察 Out[6]的输出结果，rolling_mean 列的前 5 天没有数据，第一个 MA10 是计算第一个交易日收盘价到第 10 个交易日收盘价之和再除以 10（10 天收盘价的平均值），并记录到索引标签是 2016-1-11 的记录中，然后依此向下类推，计算 MA10。

还可以通过图示将 MA10 及收盘价直观地表达出来。

```
In [7]: data.plot(style=['-', '--'])
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x7fcdfd0dc908>
```

（输出结果如图 2-10-3 所示）

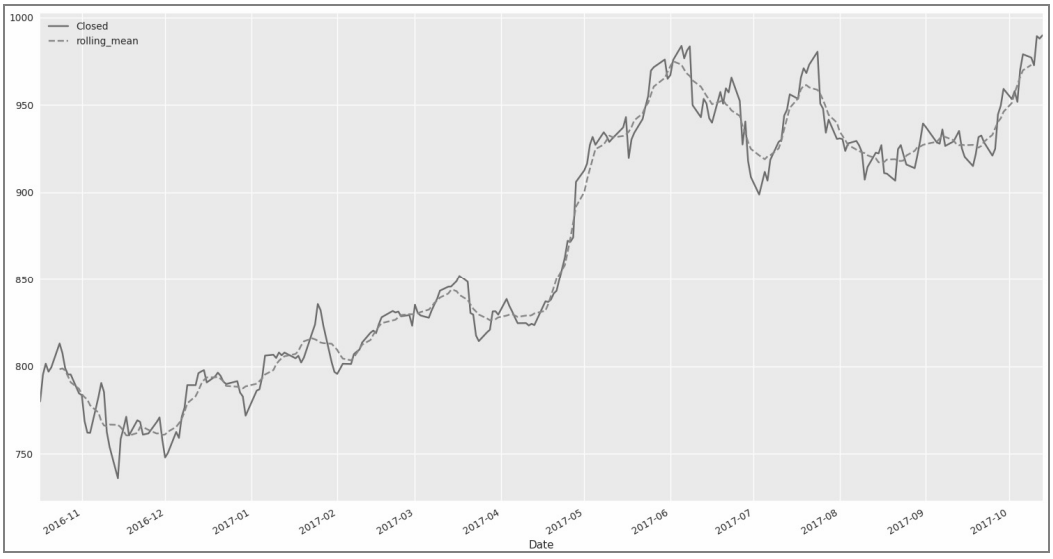


图 2-10-3　MA10 和每日收盘价

数据对象的 rolling()方法通常被形象地称为“移动窗口”，以上述计算 MA10 为例说明，假设有一个时间长度为 10 天的方框（窗口），当方框最左边在第一天的时候，方框里面有 10 天的数据（10 个收盘价），这些数据求平均，就是第一个 MA10 的值；然后将方框（窗口）向后移动一天，方框（窗口）里面又是一组 10 天的数据，再计算第二个 MA10 的值；按照这种方式将“窗口”依次移动，计算 MA10 的值，最终得到“均线”数据。

其实，“移动窗口”也可以看作是一种形式的分组运算。如果读者有兴趣，可以从分组运算的角度完成类似“移动窗口”的操作。

这里以股票数据为例，并且增加了一点图示。读者可能已经感觉到，用可视化的方式展示数据比单纯地看数字要舒服多了，所以很有必要对绘图进行完整阐述。

针对股票数据的分析，此处的示例仅仅是一个引子，后面还会对其进行学习，读者会在将来看到“量化分析”的若干影子。

第3章

数据可视化

读者学习了 NumPy 和 Pandas 之后，或许会感觉它们所包含的内容太多、太抽象了，难以掌握。对于初学者而言，有此感觉非常正常。此时，需要“振作疲惫的精神”，找一碗鸡汤，给自己“补一补”。

幸亏，从本章开始，要有一些图了——这些图依然只有懂的人才能看出其中的美，它们就是数据可视化的结果。

数据可视化是数据分析中比较庞大的一个门类，所使用的工具也比较多。比如本书中要重点介绍的 Matplotlib 就是一个很流行的本地工具。此外，还可以通过网页实现数据可视化，比如目前很流行的前端插件 echarts。本书谈及的数据可视化都是基于本地工具进行的，而本地工具除 Matplotlib 外，还有很多其他的，本章内容不会面面俱到，但会以 Matplotlib 这个应用最广的工具为例，向读者介绍基本的、常用的数据可视化方法。读者掌握本章所述内容之后，可以将此能力轻易地迁移到其他工具中。

那首歌还没有播放完，“远方也许尽是坎坷路，也许要孤孤单单走一程。早就习惯一个人，少人关心少人问，就算无人为我付青春，至少我还保有一份真”。

3.1 Matplotlib 概览

John D. Hunter 在 2003 年发布了 Matplotlib 的 0.1 版之后，Matplotlib 这个专门为 Python 提供绘图功能的第三方库逐渐得到广泛应用。截止到本书写作为止，官方网站（http://matplotlib.org/）上发布的最新版本是 2.1.0。

Matplotlib 的初心就是要为 Python 提供一个能够绘图的工具，使用方法有点类似于传统的数据工具软件 MATLAB。此外，Matplotlib 毕竟还是 Python 大家庭中的一员，它继承了 Python 的优良传统和一贯作风——免费、开源、跨平台。

随着技术的进步和时代的变迁，现在能够实现数据可视化的工具越来越多，它们都意欲向 Matplotlib 发起挑战。尽管如此，Matplotlib 的江湖地位依然稳固，并且有很多新生代也是依靠它而建立的，比如正在上升的 Seaborn——一个绘图新模块。所以，学习 Matplotlib，合算且有

必要。更何况，Matplotlib 也在与时俱进。

1．配置环境

与安装其他库一样，可以使用 pip install matplotlib 安装 Matplotlib。这个操作读者在本书第零章或许已经完成了，但是，仅有这个还不行，因为 Matplotlib 孤掌难鸣，还要有很多“依赖”。在官方文档中有专门对各种依赖的说明（http://matplotlib.org/users/installing.html），读者如果阅读该内容，不要为其所难，因为很多所谓的依赖，是在 Python 的开发中普遍适用的。笔者在这里演示安装部分依赖，以满足本章程序调试的需要。如果在以后的项目开发中需要安装新的依赖程序，可以自行根据官方文档说明安装，也可以上网搜索安装方法。当然，以下操作是在 Ubuntu 系统中进行的，虽然如此，并不意味着 Matplotlib 仅适合于此系统。Matplotlib 具有非常好的跨平台性，读者可以在所使用的任何操作系统上运行 Matplotlib，只不过笔者出于个人原因，仅仅演示在 Ubuntu 系统中的安装指令。

```
$ sudo apt-get install libpng-dev
$ sudo apt-get install libpng-dev
$ sudo apt-get install python-tk
```

如果读者觉得上述演示还不能满足需要，请参考官方文档说明，或者去“问”Google。

Matplotlib 及其依赖安装好之后，就可以使用了。但是，启用它有不同的方式。

2．执行程序绘图

要编写一段程序，并且使用 Matplotlib 绘图，就跟使用其他第三方库一样，用 import 引入即可。

```
#coding: utf-8

#filename: ./chapter03/plotpic.py

import numpy as np
import matplotlib.pyplot as plt    #①

x = np.linspace(0, 2*np.pi, 100)    #②
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1)    #③
plt.plot(x, y2)    #④

plt.show()    #⑤
```

这是一个绘制正弦和余弦函数曲线的小程序，简要说明如下。

①是常用的引入方式，用 Matplotlib 绘图，通常要使用 plt。matplotlib.pyplot 是 Matplotlib 中最常用的模块，它提供了类似 MATLAB 的接口。

②创建了自变量，作为横坐标；然后用正弦和余弦函数分别计算因变量 $y1$ 和 $y2$，作为纵坐标，这样就得到了横、纵坐标数据，此处请调用储存在头脑中的相关数学知识，有了（x, y），就可以在坐标系中描点、画曲线了。

③和④分别依据（*x*, *y*1）和（*x*, *y*2）画相应曲线，但仅仅至此还不能显示图示。注意，笔者说的是在这个程序中，跟下文在 Jupyter 中要区别开。

⑤的作用就是将③、④所画的图显示出来。一个程序文件只需要有一个 plt.show()，不论画了哪一个图，都会显示出来。

只需要草草了解上述程序即可，完整描述后续会讲。

上述程序完成后请保存，并按照已经熟悉的执行 Python 程序的方法执行此程序，命令如下。

```
chapter03$ python3 plotpic.py
```

即可得到图 3-1-1 所示的效果。

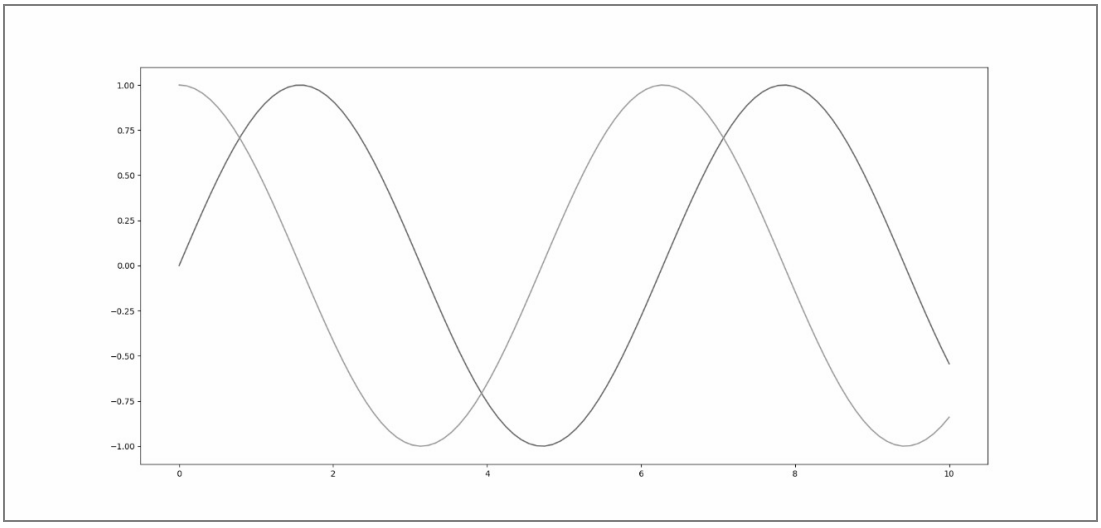


图 3-1-1　正弦和余弦曲线

3．在 Jupyter 中绘图

Jupyter 是一种交互操作模式。如果在这种模式中实现绘图，需要执行启用 Matplotlib 的操作。

```
In [1]: %matplotlib
        Using matplotlib backend: TkAgg
```

In[1]使用了魔法命令“%matplotlib”，当得到上述反馈之后，意味着在此交互模式中也可以使用 Matplotlib 了。

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        x = np.linspace(0, 2*np.pi, 100)
        y1 = np.sin(x)
        plt.plot(x, y1)
Out[2]: [<matplotlib.lines.Line2D at 0x7ff5d2a3e8d0>]
```

In[2]的代码与前面在程序文件中画图所使用的代码大致一样，区别在于此处没有使用 plt.show()。当执行 In[2]后，就能新打开一个窗口，显示如图 3-1-2 所示的正弦曲线。

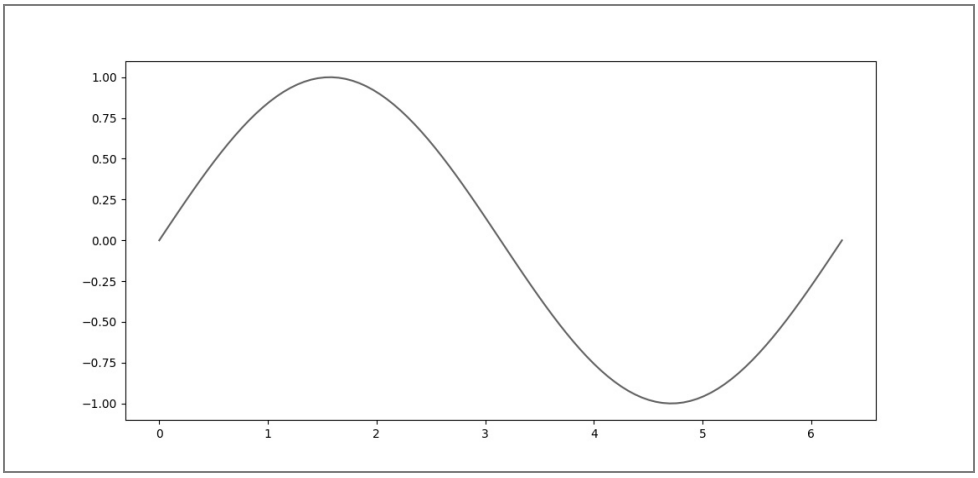


图 3-1-2　正弦曲线

不要关闭图像窗口，将激活窗口切换到交互模式中，继续输入下面的示例。

```
In [3]: y2 = np.cos(x)
        plt.plot(x, y2)
Out[3]: [<matplotlib.lines.Line2D at 0x7ff5d297bb38>]
```

再切换到图像窗口，就能看到图 3-1-3 所示的效果，其中包含正弦和余弦两条曲线，而且还自动用不同颜色区分（当然，本书不是彩色印刷，在书上看到的都是黑白色的）。

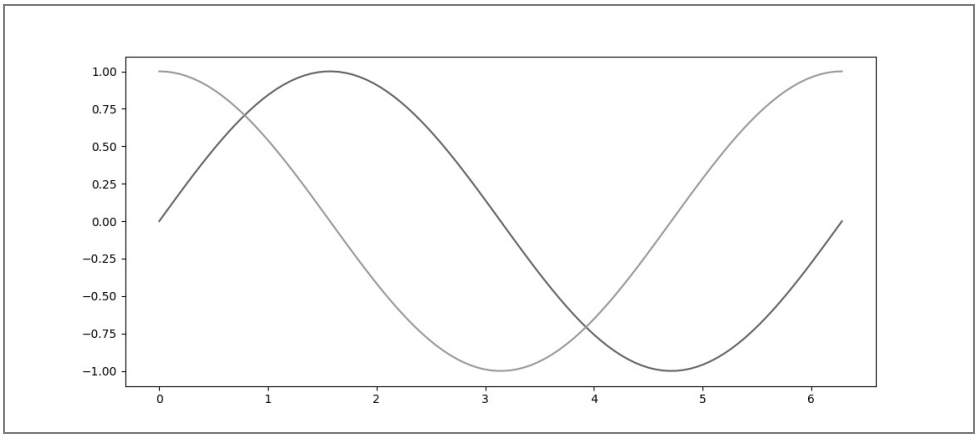


图 3-1-3　正弦函数和余弦函数图像

在后续的代码中，大多数情况下还是在 Jupyter 中调试，少数情况下会使用.py 程序文件。请读者注意在两种环境中编写程序的差异。

4．面向对象绘图

Python 中万物皆对象——这早应该是我们的共识了，不论是在 Web 开发中还是在数据分析中，都已经在不断实践这句话。那么，使用 Matplotlib 绘图当然也不例外，“绘图”及“所绘之图”都是对象。

Matplotlib 官方网站有一张著名的图，如图 3-1-4 所示。在这张图中，读者可以看到一张包

含了较完整元素的绘图。这张图就是后面研习 Matplotlib 的导图，我们对其中的各对象逐一剖析，内化为自己的知识。

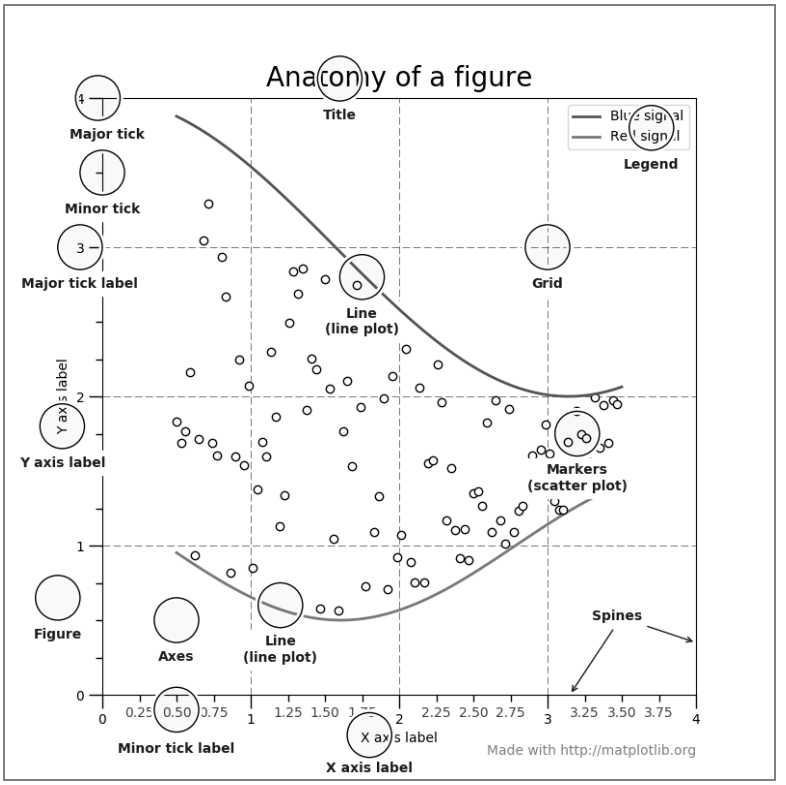


图 3-1-4　Matplotlib 图的常用元素

如果我们将这张图或绘制这张图的过程看作对象，那么图中各元素的特点就是“属性”，如何绘制某个元素就是“方法”了。下面就依照这种思想来理解 Matplotlib 的绘图过程——前提是先理解什么是面向对象，推荐参考《跟老齐学 Python：轻松入门》的相关章节。

虽然图 3-1-4 中的东西很多，但是我们有一个学习的法宝——“日拱一卒”。本着坚持不懈、永不放弃、水滴石穿的“死磕”精神，坚定信心，步步为营，每天进步一点点，就一定能攻克 Matplotlib 这个难关。

在 Matplotlib 中绘制出来的所有东西，即图 3-1-4 中的所有 Matplotlib 绘制的元素，都是 Artist 对象——“画家”的职责就是要绘制各种对象，它所能绘制的对象有两类，一类是基本元素（primitives）类，比如 Text、Line2D、Rectangle 等；另一类是容器（container）类，比如 Figure、Axes、Axis，这种对象可以再包含 Artist 的其他多个基本元素类型。网页 http://matplotlib.org/api/artist_api.html 中有一张图，包含了 Artist 对象的所有类型，图很大，不在这里展示了，请读者在合适的时机访问该网页。

一般的绘图步骤都是首先创建 Figure 对象，它类似于一张画布，可以在这张画布上绘制其他对象。

```
In [4]: fig = plt.figure()
```

此语句操作之后，在新建的窗口中将看到一个空白的画布，这就是所谓的 Figure 对象。当

然，这个对象也可以通过实例化 plt.Figure()来完成，通常习惯使用函数 plt.figure()。刚才所创建的画布对象都使用了默认值，还可以向此函数的参数提供一些数值，创建一张符合某种规范的画布。

```
plt.figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None, frameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, **kwargs)
```

建议读者通过帮助文档查看上述各个参数的含义。

按照面向对象的思路进行下去，接下来就要使用变量 fig 所引用的 Figure 对象的方法，实现在画布上画图的操作（“方法”：该对象能够做什么）。

执行完上面的语句之后，处于激活状态的是画布窗口，现在要切换到 Jupyter，但画布的图像窗口不要关闭。

```
In [5]: ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
```

add_axes()是 Figure 实例对象的一个方法。这个实例对象还有很多别的方法，读者可以自行查看，以后或许会用到一些，比如 fig.savefig()等。

先看操作结果，再解释说明。

切换到图像窗口，会看到如图 3-1-5 所示的效果（图 3-1-5 中的 Figure 和 Axes 标注，以及文字上方的圆圈，都是为了说明图中的元素而加注的，读者调试代码得到的图中没有这些）。

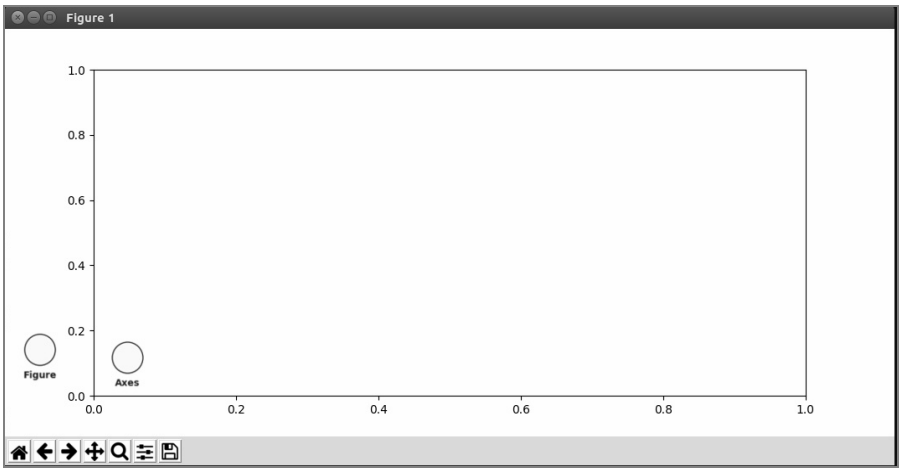


图 3-1-5　在画布上绘制坐标轴

执行 fig.add_axes()方法的结果是在画布上创建了一个 Axes 对象，它也是一个 Artist 的对象——跟 Figure 对象类似，都是“容器”，即 Axes 对象可以包含其他东西。

还是按照“面向对象”的思路，调用变量 ax 所引用的 Axes 对象的方法（切换到 Jupyter）。

```
In [6]: x = np.linspace(0, 2*np.pi, 100)
        ax.plot(x, np.sin(x))
Out[6]: [<matplotlib.lines.Line2D at 0x7ff5d1a13f28>]
```

plot()是 Axes 对象的方法之一，它可以根据数据在 Axes 对象上画图。切换到图像窗口，就会看到正弦曲线已经被画出来了，如图 3-1-6 所示。

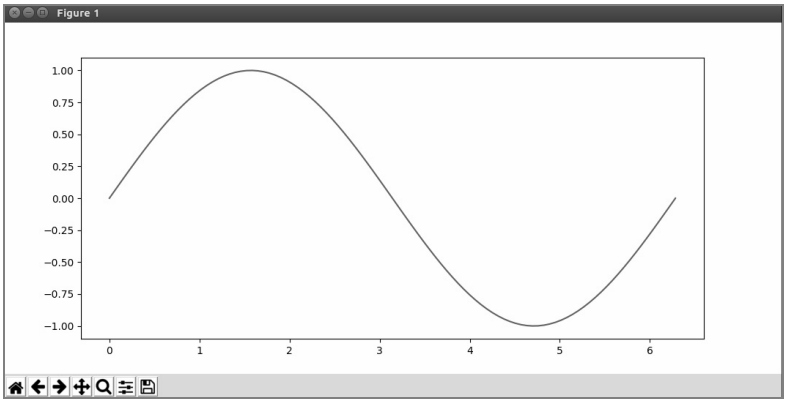


图 3-1-6　正弦曲线图像

最后，还可以调用 fig 对象的一个方法，实现将图 3-1-6 所示的内容保存到磁盘中的要求。

```
In [7]: fig.savefig("sine.png")
```

如果把上述的分解动作连贯起来，可以写到一个小程序中，然后执行。

```
In [8]: fig = plt.figure()
        ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
        x = np.linspace(0, 2*np.pi, 100)
        ax.plot(x, np.sin(x))
```

以“面向对象”的思想指导我们使用 Matplotlib 绘制图像，就是按照上述基本流程进行的。

除“面向对象绘图”外，Matplotlib 还提供另外一种绘图方式——MATLAB 风格。这是因为当初 Matplotlib 就是要让 MATLAB 用户倒戈到 Python 阵营，并且能够像使用 MATLAB 那样，所以要兼顾他们以前的操作习惯——因为“不忘初心”，所以 MATLAB 风格的操作在 Matplotlib 中一直保存着。

MATLAB 风格是什么样的？

本章的第一个绘图程序就是使用的 MATLAB 风格。

```
In [9]: plt.figure()
        plt.plot(x, np.sin(x))
```

上述代码比本章第一个绘图程序（In[2]）的代码多了一行创建 Figure 对象的语句 plt.figure()，如果没有这一句，那么 Matplotlib 会在执行 plt.plot()的时候默认创建一个“画布”。

将两种风格进行比较，会发现所谓“MATLAB 风格”就是通过 plt 操作各种绘图相关的方法和属性，而“面向对象”是通过对象来使用那些方法和属性。两者所使用的方法和属性也都是相同的或类似的，表 3-1-1 列出了几项。

表 3-1-1　plt 函数和 ax 对象方法

plt 函数	ax 对象方法
plt.plot()	ax.plot()
plt.legend()	ax.legend()
plt.xlabel()	ax.set_xlabel()
plt.ylabel()	ax.set_ylabel()

续表

plt 函数	ax 对象方法
plt.xlim()	ax.set_xlim()
plt.ylim	ax.set_ylim()
plt.title()	ax.set_title()

“面向对象”和“MATLAB 风格”没有优劣之分，完全由使用场景和个人好恶而定。所以，本书会交叉使用两者，读者也不必厚此薄彼。

在图 3-1-5 和图 3-1-6 中还可以看到一些交互操作的按钮，比如保存、移动图像等，为了显示这些按钮，截取了整个窗口。在后面的章节中，将不再显示这种类型的图示，而是直接显示绘制的结果图。但读者在调试的时候依然能看到类似图 3-1-5 的效果。

接下来，就要“细致入微”地研习每个对象了。

3.2 设置坐标系

所有的函数曲线都要画在坐标系内。这里姑且仅研习二维的正交坐标，即在数学中常见的、具有 x、y 坐标轴，并且两个坐标轴相互垂直。前面所做的图都属此类。在 Matplotlib 中，用 Axes 容器来描述数学中的坐标系。说它是一个容器，是因为它包含了坐标系中各个轴的刻度线、刻度值，以及坐标网格、坐标轴标题等——这些都可以看作 Axes 容器里面的对象。

1．坐标网格

坐标系中有网格是一件非常好的事情，可以帮助我们观察曲线的每个点的坐标数值。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        x = np.arange(0.0, 5.0, 0.02)
        y = np.exp(-x)*np.cos(2*np.pi*x)
        plt.plot(x, y)
        plt.grid(color='gray')
```

执行 In[1]后得到如图 3-2-1 所示的结果。

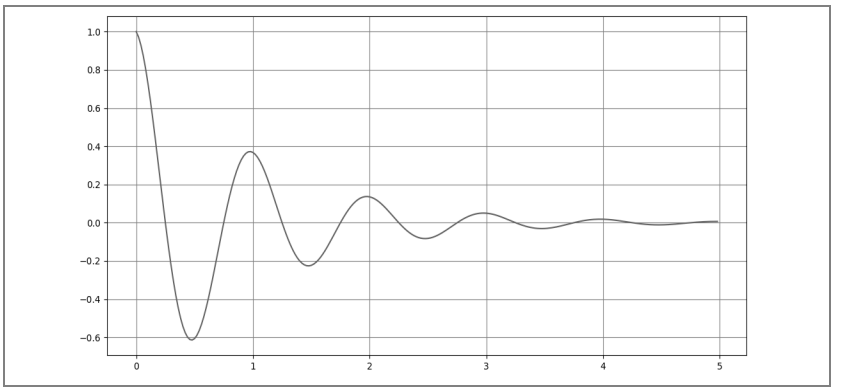


图 3-2-1　有网格的坐标系

In[1]是以“MATLAB 形式”（plt 函数）完成的绘图操作。要实现同样的操作，还可以用“面向对象”形式（ax 对象方法），如下所示：

```
In [2]: fig = plt.figure()
        ax = fig.add_axes([0.1, 0.1, 0.8,0.8])
        ax.grid(color='gray')
        ax.plot(x, y)
```

不论用哪种方式，要绘制坐标网格，都要执行 grid()方法（ax.grid()或 plt.grid()），并且两个对象的此方法参数列表一样。

类似的情形在以后的各种操作中还会出现，笔者就不再进行对比操作了，而是根据具体情况选择其中某一种操作形式作为示例，请读者特别注意。

除在上面的操作中使用 color 参数设置表格线的颜色外，还可以使用表 3-2-1 中的参数来设置表格线的其他几项基本属性。当然，grid()方法所能接受的参数不仅限于这几个，关于更多参数，请读者用“ax.grid?”方法打开官方文档并阅读。

表 3-2-1　grid()部分参数

参　数	说　明
axis	默认 axis='both'，还可以设置为'x'或者'y'，分别表示表格线是垂直于 *X* 轴还是垂直于 *Y* 轴
color	设置表格线的颜色
linestyle	设置表格线的线形，例如 linestyle='-'
linewidth	设置表格线的宽度，例如 linewidth=2

2．坐标轴

给 In[2]的程序增加两行，目的是对图像的坐标轴进行重新设置。

```
In [3]: fig = plt.figure()
        ax = fig.add_axes([0.1, 0.1, 0.8,0.8])
        ax.grid(color='gray')
        ax.plot(x, y)
        ax.set_xlabel("x axis")     #新增，为 X 轴设置标题
        ax.set_xlim((-2, 10))    #新增，为 X 轴设置数值范围
Out[3]: (-2, 10)
```

ax.set_xlabel("x axis")的作用是在坐标系的横轴下面显示该坐标轴的标题为“x axis”，同理，也可以使用 ax.set_ylabel()为纵轴设置坐标轴标题。

ax.set_xlim((-2, 10))的作用是设置横轴的刻度显示范围，注意 ax.set_xlim()以元组(-2, 10)为参数，表示坐标轴最左侧显示的刻度值是-2，最右侧显示的是刻度值 10，即最小值和最大值。set_xlim()与前述所用过的方法稍有不同，它有返回值——返回了坐标轴的刻度值范围。同样，也可以用 ax.set_ylim()来设置纵轴。

执行完以上两个指令之后，再切换到图像窗口，会看到如图 3-2-2 所示的效果。请对比图 3-2-2 和图 3-2-1，并仔细观察我们欲修改的部位是否已经按照要求修改了（图中横坐标的标题“x axis”字号比较小，请注意观察）。请不要关闭图 3-2-2 所示的窗口，而是继续 In[4]的操作。

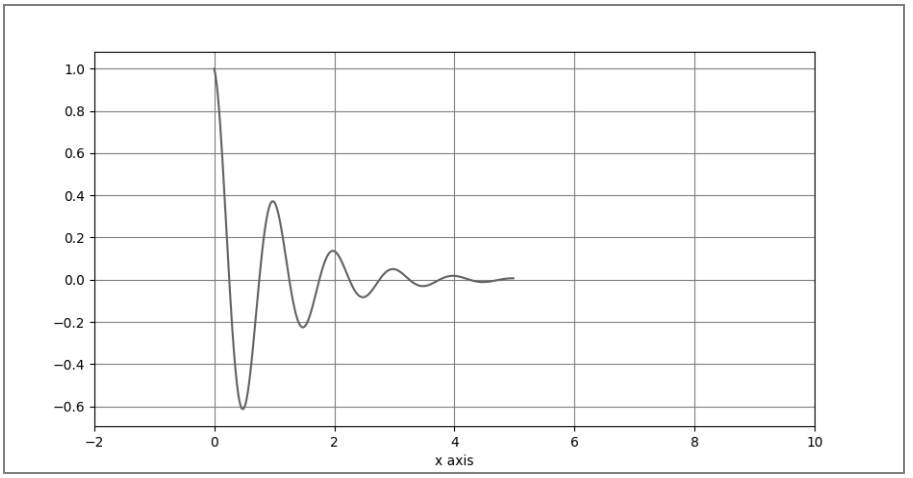


图 3-2-2　坐标轴标题和数值范围

图 3-2-2 的有效图示部分所占比例很小，也就是利用率太低了。根源就是横轴设置的范围不合理，还是恢复到 0~5 的范围比较好。同时，还希望显示的刻度为[0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5.]，于是可以像下面这样操作以实现需求。

```
In [4]: fig = plt.figure()
        ax = fig.add_axes([0.1, 0.1, 0.8,0.8])
        ax.grid(color='gray')
        ax.plot(x, y)
        ax.set_xlabel("x axis")
        ax.set_xlim((0, 5))      #X 轴数值刻度范围恢复为 0~5
        ax.set_xticks(np.linspace(0, 5, 11))     #或者 plt.xticks(np.linspace(0, 5, 11))
Out[4]: ([<matplotlib.axis.XTick at 0x7ff53923e278>,
          <matplotlib.axis.XTick at 0x7ff539241dd8>,
          <matplotlib.axis.XTick at 0x7ff5392694e0>,
          <matplotlib.axis.XTick at 0x7ff539269ef0>,
          <matplotlib.axis.XTick at 0x7ff5391ee748>,
          <matplotlib.axis.XTick at 0x7ff539269898>,
          <matplotlib.axis.XTick at 0x7ff5392302e8>,
          <matplotlib.axis.XTick at 0x7ff539230c18>,
          <matplotlib.axis.XTick at 0x7ff539237cf8>,
          <matplotlib.axis.XTick at 0x7ff5391d11d0>,
          <matplotlib.axis.XTick at 0x7ff5391d19e8>],
         <a list of 11 Text xticklabel objects>)
```

输出结果如图 3-2-3 所示，不仅利用率高了，而且横轴显示的数值精确度也提高了（是不是小数点后的位数越多就越准确呢？读者可以思考）。

以上对坐标轴的操作，只是雕虫小技，更丰富多彩的操作还在后面。

先画一张由随机数据得到的图，如图 3-2-4 所示。

```
In [5]: ax = plt.axes()
        ax.plot(np.random.rand(50))
        ax.yaxis.set_major_locator(plt.NullLocator())
        ax.xaxis.set_major_formatter(plt.NullFormatter())
```

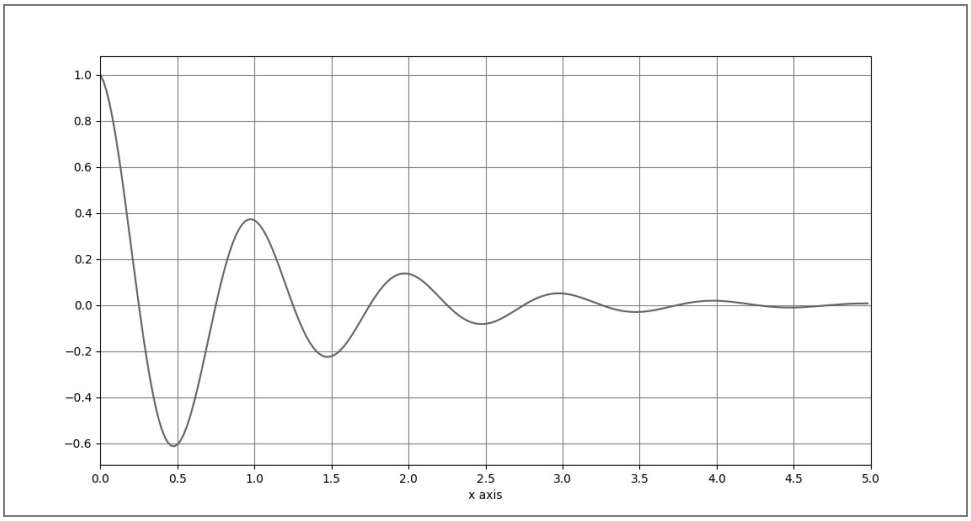


图 3-2-3　修改坐标轴刻度

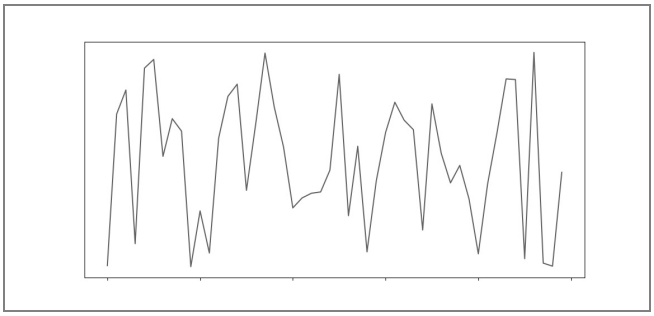


图 3-2-4　控制坐标轴刻度

这里新建了一个 Axes 容器对象。重点关注 ax.yaxis.set_major_locator(plt.NullLocator())和 ax.xaxis.set_major_formatter(plt.NullFormatter())的操作结果，将代码和结果图对照更容易理解。

坐标轴的刻度由短线和某些短线附近的文本组成，为了便于表述，在这里将短线命名为“刻线”，把短线附近的文本命名为“标示”，在其他资料中或许有别的名称，本书中笔者用这种命名方法——在 Matplotlib 中有相应的对象描述。

- locator 对象描述刻度的“刻线”，当执行 ax.yaxis.set_major_locator(plt.NullLocator())之后，刻线不存在了，即使有“标示”也没有意义了，正所谓“皮之不存，毛将焉附”。
- formatter 对象描述刻度的“标示”，所以 ax.xaxis.set_major_formatter(plt.NullFormatter())之后 *X* 轴没有了标示，只有刻线了——刻线可以单独存在，类似于刻度尺，在刻度尺上不是每条刻线都有标示。

提示读者特别关注的那两个语句中，所涉及的方法 set_major_locator()和 set_major_formatter()分别用来设置坐标轴对象的“刻线”和“标示”。

在理解上述操作之后，我们要做一件看起来很酷的事情。请读者依照如下操作，并且在浏览代码的时候尽可能理解其含义。因为做很酷的事情要用到一些数据，所以先安装一个在机器学习中用途很广的库——scikit-learn。

```
$ sudo pip3 install -U scikit-learn    #如果安装有问题，请上网搜索解决方法
```

scikit-learn 中有一个数据集，里面包含很多人的面部图像，在下面的程序中将要使用这些数据。

```
In [6]: from sklearn.datasets import fetch_olivetti_faces
        faces = fetch_olivetti_faces().images
        fig, ax = plt.subplots(5, 5, figsize=(5, 5))    #①
        fig.subplots_adjust(hspace=0, wspace=0)    #②
        for i in range(5):
            for j in range(5):
               ax[i, j].xaxis.set_major_locator(plt.NullLocator())
               ax[i, j].yaxis.set_major_locator(plt.NullLocator())
               ax[i, j].imshow(faces[10*i+j], cmap='bone')    #③
```

先不解释代码，直接看运行效果，如图 3-2-5 所示。

图 3-2-5　5×5 的子图

图 3-2-5 不是一个坐标系，而是由 5×5 个坐标系组成的，每个坐标系内只显示一个人脸图像。如何一下子生成这么多坐标系？语句①中的 subplots()专司此事，从函数名称中就可以推测，它的作用就是将画布 Figure 分成若干个区。

```
In [7]: fig, ax = plt.subplots(2, 1)
```

图 3-2-6 是 In[7]的执行结果，如果以(2, 1)为 subplots()的参数，则在画布上按照“0 轴方向为 2、1 轴方向为 1”的布局生成 2×1 个分区，每个分区就是我们熟悉的 Axes 容器对象（看完之后不要关闭图 3-2-6 所示的窗口）。

subplots()的返回值有两个，一个是 Figure 对象，另外一个是 Axes 容器对象。如 In[7]，变量 ax 就引用了两个分区的 Axes 容器。

```
In [8]: ax
Out[8]: array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f783ef9b2e8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f783efe6908>],
        dtype=object)
```

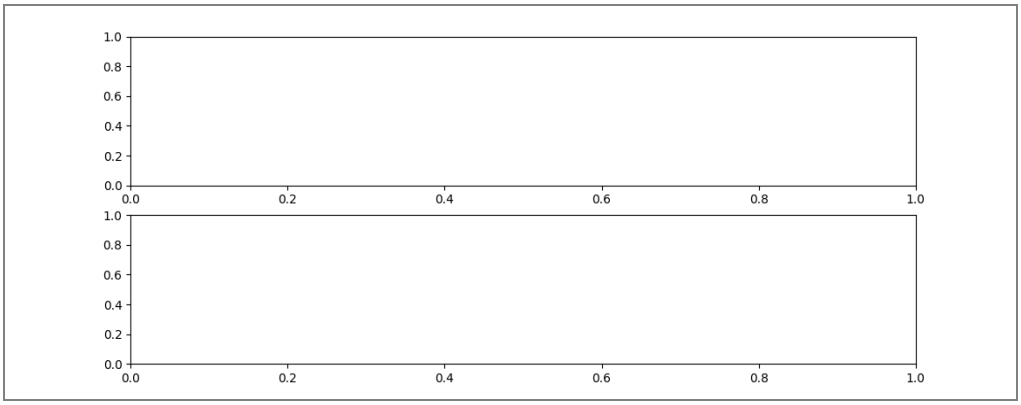


图 3-2-6　subplots()生成的子图

可以通过索引得到其中某个分区的 Axes 对象，比如：

```
In [9]: ax[0].plot(np.linspace(0, 2*np.pi, 10), np.sin(np.linspace(0, 2*np.pi, 10)))
```

于是就在第一个分区的 Axes 对象中绘制了正弦曲线（如图 3-2-7 所示）。因为所取的坐标数量有限，所以曲线不光滑。

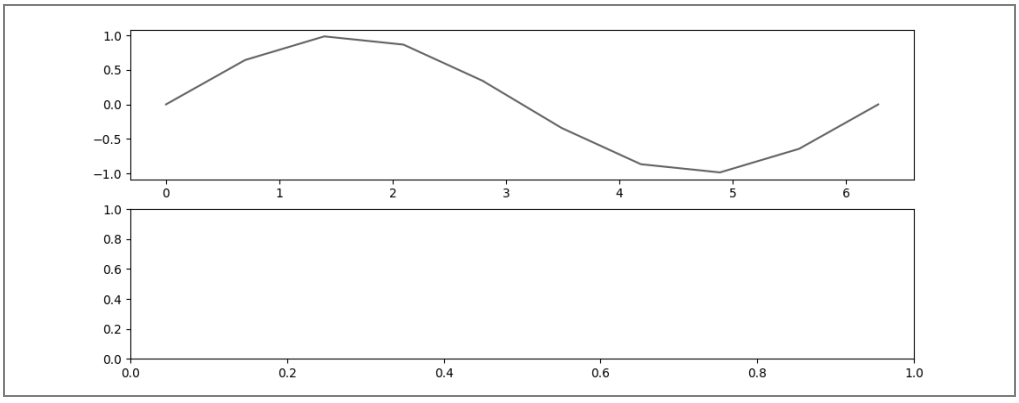


图 3-2-7　ax[0]子图中绘制正弦曲线

回头再看 In[6]中的 for 循环，就是通过依次调用每个分区的 Axes 对象，完成对其的绘图操作的。

In[6]中语句②的 subplots_adjust()函数的作用是调整分区之间的相对位置，它的完整参数列表为：

```
subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)
```

其中，wspace 和 hspace 分别表示分区之间空隙的宽和高。

In[6]中语句③的 imshow()函数名称如果写完整，应该是“image show”，读者由此肯定知晓其含义了，它的功能就是在坐标系中显示图片。建议用“ax.imshow?”方式查看其文档。

在 In[6]的程序中，我们对于是否显示坐标轴的“刻线”和“标示”加以控制。其实对它们，还可以做更复杂的一些操作，下面看一个复杂一点的完整程序。

```
#coding: utf-8
```

```
import numpy as np
import matplotlib.pyplot as plt

from matplotlib.ticker import MultipleLocator, FormatStrFormatter

t = np.linspace(0, 100, 100)
s = 9.8 * np.power(t, 2) / 2
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(t, s)

ax.set_ylabel('displacement')
ax.set_xlim(0, 100)
ax.set_xlabel('time')

xmajor_locator   = MultipleLocator(20)    #①
xmajor_formatter = FormatStrFormatter('%1.1f')    #②
xminor_locator   = MultipleLocator(5)

ymajor_locator   = MultipleLocator(10000)
ymajor_formatter = FormatStrFormatter('%1.1f')
yminor_locator   = MultipleLocator(5000)

ax.xaxis.set_major_locator(xmajor_locator)    #③
ax.xaxis.set_major_formatter(xmajor_formatter)    #④

ax.yaxis.set_major_locator(ymajor_locator)
ax.yaxis.set_major_formatter(ymajor_formatter)

ax.xaxis.set_minor_locator(xminor_locator)    #⑤
ax.yaxis.set_minor_locator(yminor_locator)

ax.grid(True, which='major')
ax.grid(True, which='minor')

for tick in ax.xaxis.get_major_ticks():    #⑥
    tick.label1.set_fontsize(16)    #⑦

plt.show()
```

程序虽然有点长，但只要有耐心，就一定能搞懂。

模块 matplotlib.ticker 包含了一系列与坐标轴刻度——“刻线”（Locator）和“标示”（Formatter）——相关的方法，借此模块中的方法可以实现多样化的刻度绘制（完整的官方文档见 http://matplotlib.org/api/ticker_api.html）。

在解释代码之前，先理解坐标轴刻度可以分“主”“次”的含义。如图 3-2-8 所示的刻度尺，“刻线”较长并且有“标示”的被称为“主刻度”，另外的则为“次刻度”。

这种标注刻度值的方法是聪明的，既避免了刻度标注过多而显得杂乱，又显示了刻度尺的测量精确度，这种聪明的方式在 Matplotlib 绘图中也采用了。

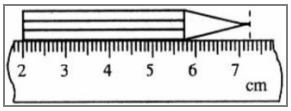


图 3-2-8　刻度尺

“from matplotlib.ticker import MultipleLocator, FormatStrFormatter”所引入的 MultipleLocator 类就是用于设置刻度的倍数的，例如①中的含义就是 *X* 轴的“主刻线”是 20 的倍数，即对于本程序而言，*X* 轴上每 20 是一个“主刻线”。FormatStrFormatter 用于设置“标示”的显示格式，例如②中规定了“主刻度”的“标示”的显示格式为小数点后要有 1 位小数。

①和②所建立的对象分别作为参数用于③和④。ax.xaxis 得到的是 *X* 轴 Axes 对象，它具有以下两个方法：set_major_locator()设置“主刻线”；set_major_formatter()设置相应的“标示”。

⑤中的 set_minor_locator()设置“次刻线”，因为“次刻线”不需要标示，所以就没有使用 set_minor_formatter()函数，读者也可以尝试使用。

X 轴的刻度就设置好了。用同样的方法可以设置 *Y* 轴的刻度。

最后，⑥通过循环 ax.xaxis.get_major_ticks()得到 *X* 轴的主刻度，然后在⑦中对每个主刻度的“标示”字号进行设置，比默认字号大一些。

笔者的解释完毕，其他语句读者应该能够理解。最后请读者逐行阅读代码，再对照图 3-2-9 所示的显示效果。

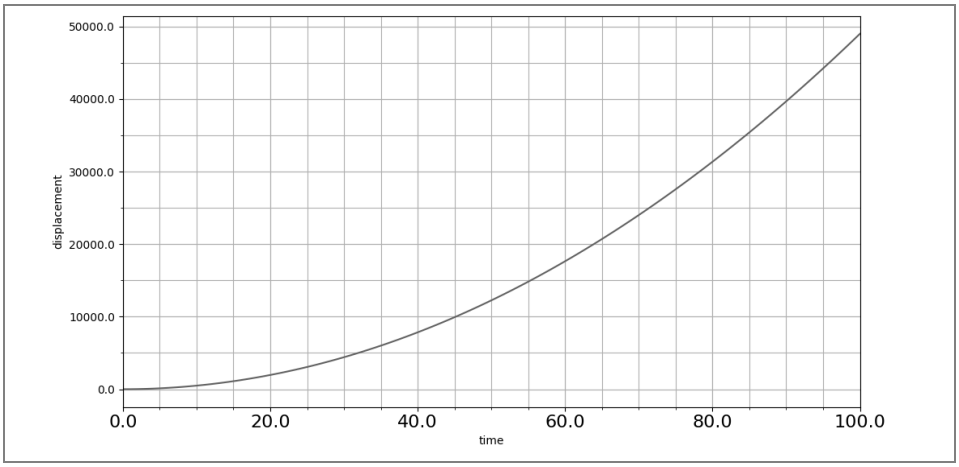


图 3-2-9　$s=g*t^2/2$ 的图像

以上示例并没有将坐标轴的所有操作都涵盖。读者在 Jupyter 中通过 ax.xaxis.<TAB>可以看到 *X* 轴 Axes 实例对象下的各个方法。特别建议浏览一番，在名称上混个“脸熟”。在后期项目中如果用到，可以详细查看其文档。

这样，一个坐标系就设置完毕了，但有时候在一张画布上要画多个坐标系。

3．分区

如果使用 plt.plot()绘图，则画布上仅有一个坐标系；而如果使用 plt.subplots()绘图，则可以在画布上生成多个坐标系，每个坐标系都是一个 Axes 容器对象。

```
In [10]: plt.subplots?

         Signature: plt.subplots(nrows=1, ncols=1, sharex=False, sharey=False,
                                 squeeze=True, subplot_kw=None, gridspec_kw=None,
                                 **fig_kw)
         Docstring:
         Create a figure and a set of subplots

In [11]: plt.subplot?

         Signature: plt.subplot(*args, **kwargs)
         Docstring:
         Return a subplot axes positioned by the given grid definition.
         Typical call signature::
             subplot(nrows, ncols, plot_number)
```

subplots()和 subplot()都是 plt 中的函数，请读者特别注意区分它们的不同。

```
In [12]: fig, ax = plt.subplots(3, 3, sharex='col', sharey='row')
```

In[12]的操作之后，除能够看到图 3-2-10 所示的 3×3 个分区外，还能得到 Figure()对象和 ax 引用的 Axes 容器对象（看到此图后，不要关闭窗口，继续后续操作）。

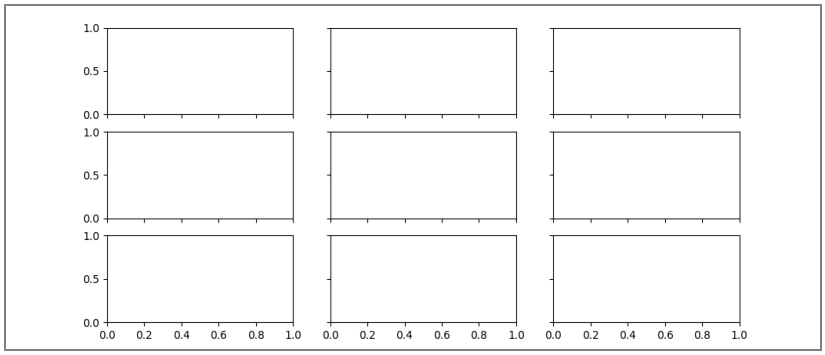


图 3-2-10　subplots()生成的分区

```
In [13]: ax.shape
Out[13]: (3, 3)

In [14]: ax
Out[14]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f783e5d70f0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7f783eb38278>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7f784955ecc0>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x7f783e34c780>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7f783e3b1a90>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7f783efa00f0>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x7f783d64a438>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7f783d8dba90>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7f783f023be0>]],
         dtype=object)
```

正如图 3-2-10 所示，ax 所引用的对象不是一个 Axes 容器，而是由若干个 AxesSubplot 对象（也是 Axes 容器对象）组成的数据。后面就可以根据数组的知识，使用每个 AxesSubplot 容器

对象。

```
In [15]: for i in range(3):
             for j in range(3):
                 ax[i, j].text(0.5, 0.5, str((i, j)), fontsize=18, ha='center')
```

以循环语句，通过 ax[i, j]得到每个分区的 Axes 对象，text()是 Axes 容器对象的方法，能够向该容器中增加文本内容。

```
text(x, y, s, fontdict=None, withdash=False, **kwargs)
```

以参数列表中的 *x* 和 *y* 的值说明文本 *s* 在坐标系内的位置，规定当 *x*=0、*y*=0 时的位置是左下角；当 *x*=1、*y*=1 时的位置是右上角。

如果得到图 3-2-10 之后，并没有关闭图像窗口，执行了 In[15]中的语句，就会在图像窗口中看到如图 3-2-11 所示的效果（如果不小心关闭窗口也没关系，重新执行 In[12]，再执行 In[15]即可）。

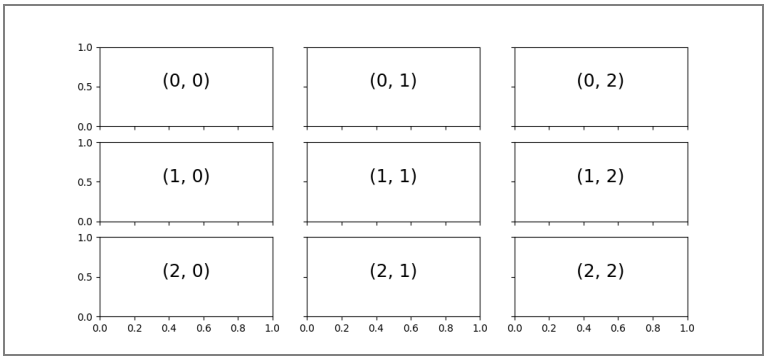


图 3-2-11　操作每个子图的效果

从 In[11]的文档中可知，plt.subplot()的作用是得到分区中的一个子图（注意拼写，没有后缀 s），如果要将它应用在分区中，可以通过循环语句来完成。

```
In [16]: for i in range(1, 7):
             plt.subplot(2, 3, i)
             plt.text(0.5, 0.5, str((2, 3, i)), fontsize=18, ha='center')
```

plt.subplot(2, 3, i)函数在画布上以 row=2、col=3 定义了分区的分布，然后通过循环依次访问每个区域，并创建坐标系，如图 3-2-12 所示。注意子图编号，居然不是从 0 开始的。

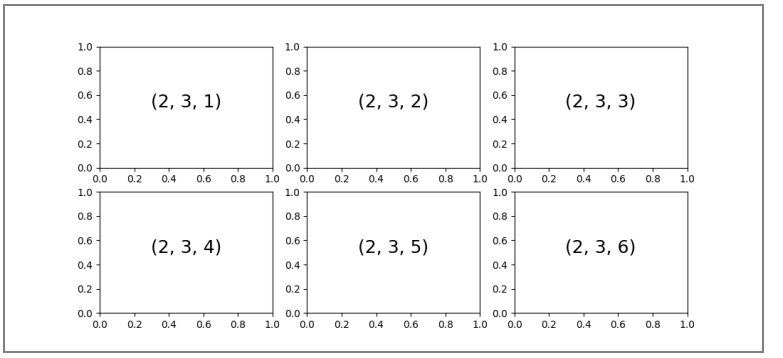


图 3-2-12　plt.subplot()生成子图

In[16]是用 MATLAB 风格编写的，如果“面向对象”，则要注意函数名称稍有不同。

```
In [17]: fig = plt.figure()
         fig.subplots_adjust(hspace=0.4, wspace=0.4)
         for i in range(1, 7):
             ax = fig.add_subplot(2, 3, i)
             ax.text(0.5, 0.5, str((2, 3, i)), fontsize=18, ha='center')
```

In[17]中使用了 Figure 对象的 add_subplot()方法增加分区，效果与 plt.subplot()效果等同。在上述代码中，还有一句 fig.subplots_adjust(hspace=0.4, wspace=0.4)，作用是设置相邻分区的间距，其中参数 hspace 和 wspace 的值表示其间距为分区长或宽的 40%。

还记得前面使用过的 fig.add_axes()方法吗？这个方法也可以在同一张画布（Figure 对象）上创建多个 Axes 对象——等同于坐标系。

```
In [18]: fig = plt.figure()
         ax1 = fig.add_axes([0.1, 0.1, 0.8, 0.8])
         ax2 = fig.add_axes([0.6, 0.5, 0.2, 0.3])
```

fig.add_axes([0.1, 0.1, 0.8, 0.8])内的参数是按照[left, bottom, width, height]顺序确定 Axes 对象的位置和大小的，left 和 bottom 分别表示距离画布左侧和底部的距离，其数值为画布宽和高的百分比，比如 left=0.6，表示距离画布左边缘的距离为画布宽度的 60%；width 和 height 分别表示子图的宽和高，width=0.2 表示子图宽度为画布宽度的 20%（如图 3-2-13 所示）。

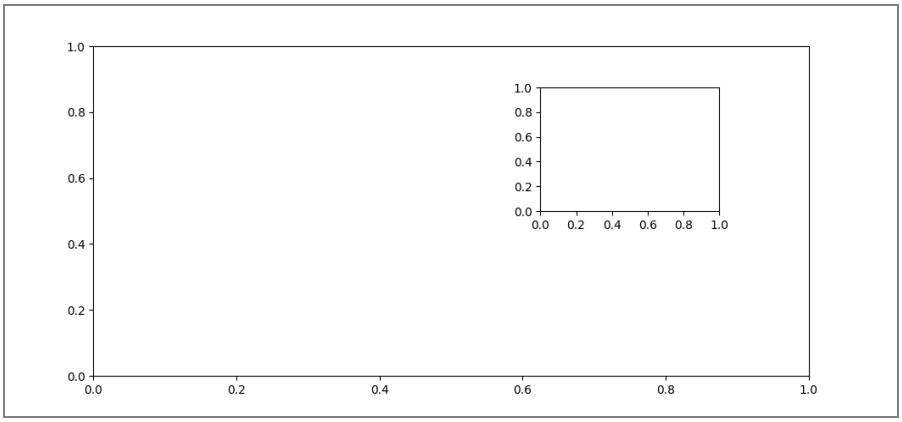


图 3-2-13　add._axes()生成的坐标系

也可以使用 MATLAB 风格编写，只要使用 plt.axes()函数，就能实现与 In[18]相同的效果。

还有一个更灵活的创建分区的方法需要介绍，若没有这个方法，分区的布局就会不灵活或者实现灵活布局就会有点麻烦。使用这个方法可以灵活地实现各种布局。

```
In [19]: g = plt.GridSpec(3, 3, hspace=0.3, wspace=0.3)
         plt.subplot(g[0, 0])    #①
         plt.subplot(g[:2, 1:])    #②
         plt.subplot(g[1:, 0])    #③
         plt.subplot(g[2, 1:])    #④
```

plt.GridSpec(3, 3, hspace=0.3, wspace=0.3)的作用是将所创建的画布（Figure 对象）划分为 3×3 的网格，如图 3-2-14 所示，分别根据图中标识的区域创建大小不一的子图。

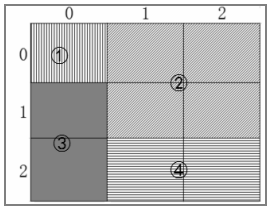


图 3-2-14　大小不同的子图

依然使用 plt.subplot()函数，注意所传入的参数是使用变量 g 所引用的对象的切片，类似于多维数组的切片，代码中的①②③④分别与图 3-2-14 中标号的区域对应。如图 3-2-15 所示为 In[19]的执行结果，请与图 3-2-14 及代码对照，理解 In[19]的含义。

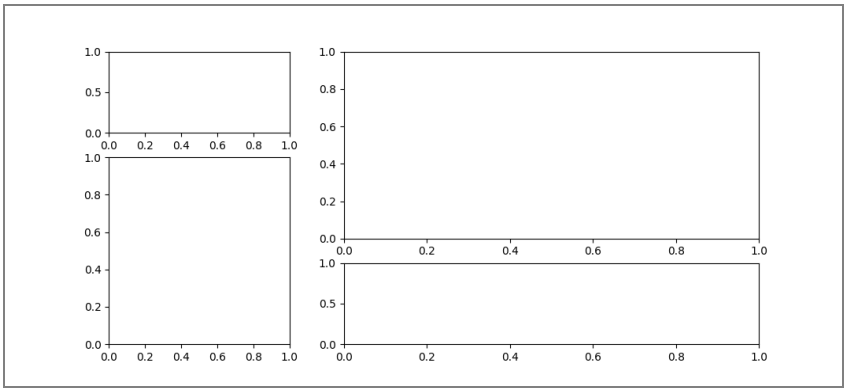


图 3-2-15　借助 GridSpec()划分区域

综上所述，绘图的过程可以概括为：

① 创建 Figure 对象——相当于一张画布；

② 用 subplots()等函数创建分区——将画布分为若干个部分，也可以是一个部分；

③ 在执行上一步的同时创建了 Axes 容器对象——相当于同时在分区上创建坐标系；

④ Axes 对象——坐标系的各个组成部分，可以被单独操作、设置；

⑤ 在 Axes 容器对象内绘图——这是下一节要重点讲述的。

这样，我们已经完成了“布局”的工作，接下来要进入“绘制图像”环节了。

3.3　绘制图像

在坐标系里面绘制的内容，我们笼统地称之为图像，它包括各种曲线图、散点图和各种统计图等，还可以包括图片，比如前面已经操作过的人脸图片。对于各种图像，在 Matplotlib 中都提供了相应的方法进行绘制。本节介绍基本的曲线和统计图表的绘制。

1．曲线

在坐标系内，如果知道了 X 和 Y 之间的对应关系——函数，就可以画出此函数的图像，即函数曲线——直线是一种特殊的曲线。

用 Matplotlib 画函数曲线，读者应该不陌生，因为前面已经介绍过 plt.plot()函数。此处我们要研究的是它的参数，通过对参数的设置，让曲线更“多姿多彩”，如图 3-3-1 所示。

五彩世界让人迷恋，所以先看线的色——既然本书是黑白印刷的，那么读者就一定要调试才能欣赏了。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
In [2]: x = np.linspace(0, 2*np.pi, 100)
        plt.plot(x, np.sin(x), color='blue')
        plt.plot(x, np.sin(x-np.pi/3), color='r')
        plt.plot(x, np.sin(x-2*np.pi/3), color='#A52A2A')
```

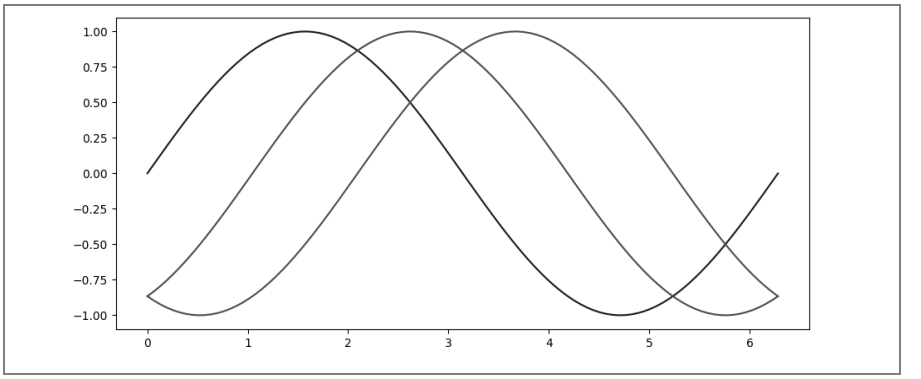


图 3-3-1　不同色彩的曲线（具体见调试结果）

plot() 函数用参数 color 设置曲线的颜色。在 Matplotlib 中，不止这个函数中有颜色的设置，所以有必要对颜色做统一规定。但凡以如下任何一种方式指定颜色，都是能被 Matplotlib 中的函数（方法）所理解和接受的。

- RGB 或者 RGBA 模式，由[0, 1]之间的浮点数组成的元组表示，比如(0.1, 0.2, 0.5)或(0.1, 0.2, 0.5, 0.3)。
- RGB 或者 RGBA 模式，由十六进制的字符串表示，比如'#0F0F0F'或'#0F0F0F0F'。
- 灰度方式，由[0, 1]之间的浮点数的字符串表示，比如'0.5'。
- {'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}之一，Matplotlib 所认定的纯色简称，或者使用全称（与简称对应，依次是：blue、green、red、cyan、magenta、yellow、black、white）。
- X11/CSS4 颜色名称（X11 为颜色名称，请参阅 https://en.wikipedia.org/wiki/X11_color_names）。
- XKCD 颜色名称，在使用的时候要以'xkcd:'为前缀，例如'xkcd:sky blue'（关于 XKCD，请参阅 https://xkcd.com/color/rgb/）。
- T10 调色板的 Tableau 颜色，{'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple', 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan'}，这是默认的循环配色。
- 用“CN”缩略形式表示，C 后面紧跟一个整数（N 是一个整数），这个整数是 matplotlib.rcParams['axes.prop_cycle']所定义的循环配色中的索引。

Matplotlib 2.0 对颜色的默认配置做了较大调整，将默认的循环配色由['b', 'g', 'r', 'c', 'm', 'y', 'k']变更为颜色调色板 category10，并且新的默认颜色值仅能通过十六进制来指定。如果要回到 1.x 的默认设置，可以通过如下操作完成。

```
In [3]: import matplotlib
        matplotlib.style.use('classic')    #回到 1.x 的默认样式
        from cycler import cycler    #恢复旧的循环配色
        matplotlib.rcParams['axes.prop_cycle'] = cycler(color='bgrcmyk')
```

个人认为，没有必要复古。

不论是在哪一个方法或函数中，只要用到颜色，都可以使用 color 参数，引用上述某一种方式所表达的色彩值即可。

下面这段程序的目的是为了展示 Matplotlib 对色彩的支持，读者可以权当一个练习，检验自己阅读程序的耐心和能力。建议阅读方法是逐行对下面的程序进行注释，并且进行调试——因为本书是黑白印刷的，所以建议在计算机上调试看效果，如图 3-3-2 所示。

```
#coding:utf-8

"""
filename:pltcolor.py
display the color of plt
"""
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.cm as cmx
import numpy as np

curves = [np.random.random(20) for i in range(10)]
values = range(10)

fig = plt.figure()
ax = fig.add_subplot(111)
jet = cm = plt.get_cmap('jet')
cnorm = colors.Normalize(vmin=0, vmax=values[-1])
scalar_map = cmx.ScalarMappable(norm=cnorm, cmap=jet)

lines = []
for idx in range(len(curves)):
    line = curves[idx]
    color_val = scalar_map.to_rgba(values[idx])
    color_text = ('color: (%4.2f,%4.2f,%4.2f)'%(color_val[0],color_val[1],color_val
[2]))
    ret_line, = ax.plot(line, color=color_val, label=color_text)
    lines.append(ret_line)

handles,labels = ax.get_legend_handles_labels()
ax.legend(handles, labels, loc='upper right')

plt.show()
```

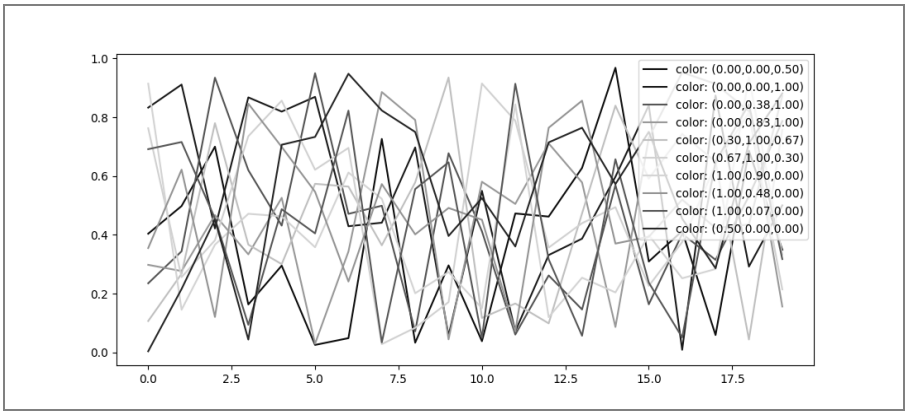


图 3-3-2　Matplotlib 中的多种色彩（建议观察调试后的效果）

对于曲线而言，色彩是一种属性。此外还有别的属性，比如线的样式，亦称“线型”，常见的有实线、虚线等（以下承接生成图 3-3-1 的程序操作）。

```
In [4]: plt.plot(x, 2*x, linestyle="dashdot")
        plt.plot(x, -2*x + 2, linestyle=":")
```

在 plot()中，可以通过 linestyle 参数来确定线的样式，如图 3-3-3 所示。不过读者要注意，Matplotlib 所支持的线的样式是有限的。表 3-3-1 中列出了所支持的几种线型，供读者参考。

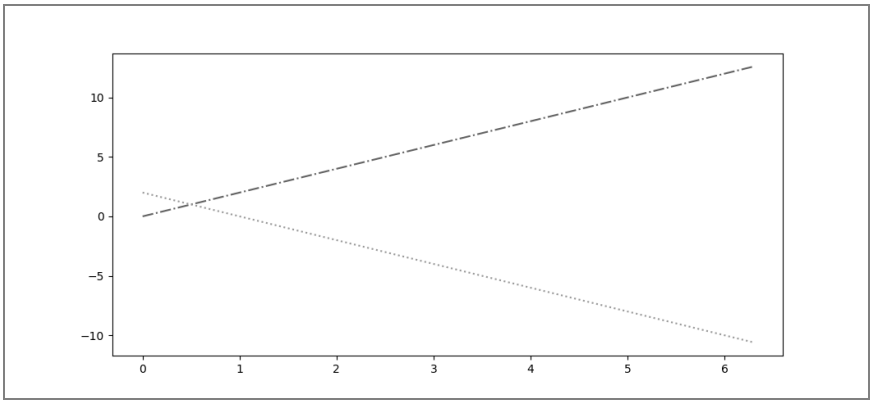


图 3-3-3　不同样式的线

表 3-3-1　常用的线型

名　　称	solid	dashed	dashdot	dotted
符号	-	--	-.	:
样式				

在 In[4]中绘制了两条曲线，如果我们不对线的色彩予以规定，Matplotlib 会根据默认的循环配色，为每条曲线自动设置颜色。当然，也可以在 plot()中再增加一个 color 参数，如图 3-3-4 所示。

```
In [5]: plt.plot(x, np.sin(x), linestyle="-.", color='blue')
        plt.plot(x, np.cos(x), linestyle=":", color='black')
```

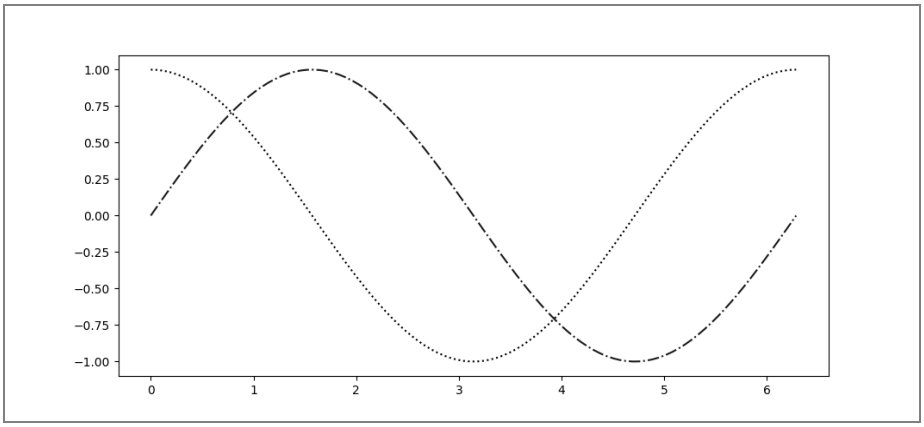


图 3-3-4　指定颜色和线型

任何曲线都是根据坐标点而来的。细心的读者在阅读 plot()文档的时候会发现，其参数列表中有一个名为 marker 的参数，它就是用来标记坐标点的，如图 3-3-5 所示。

```
In [6]: plt.plot(range(10), linestyle='--', marker='o', markersize=16, markerfacecolor
                ='b', color='r')
        plt.grid(True)
```

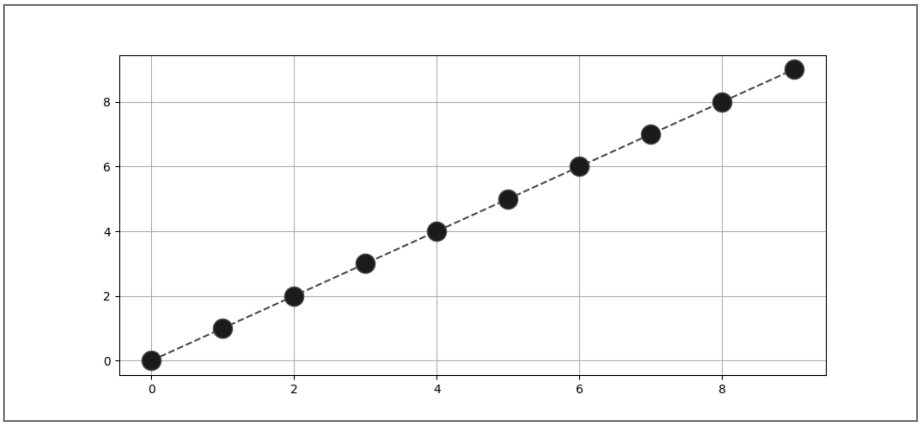


图 3-3-5　带有参数 marker 的线

可以用来标记坐标点的符号还有很多，读者在“plt.plot?”文档中可以看到很多 marker 的样式。

In[6]的 plot()函数中可以把 linestyle、marker、color 缩写成一个字符串。下面的 In[7]与 In[6]的执行效果一样。

```
In [7]: plt.plot(range(10), '--or', markersize=16, markerfacecolor='b')
        plt.grid(True)
```

再看一下 In[7]中的语句，就会产生一个新的问题，是否可以任意指定标记的点？

当然可以！如图 3-3-6 所示。

```
In [8]: plt.plot(range(10), '-Dr', markersize=16, markerfacecolor='b', markevery=
                [2,4,6])
```

```
plt.grid(True)
```

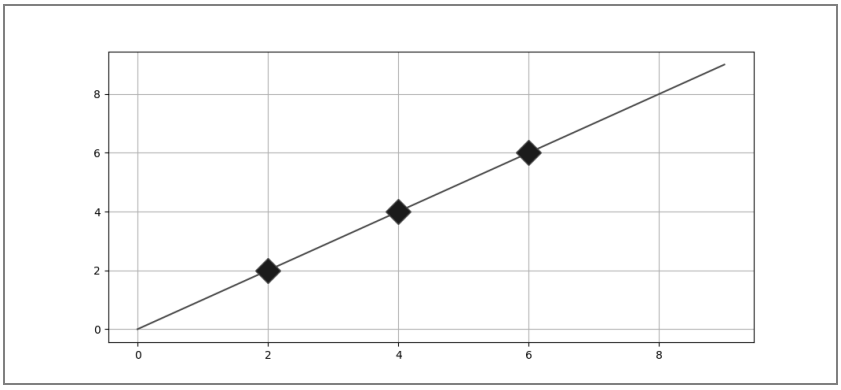


图 3-3-6　标记规定的坐标点

In[8]相比 In[7]，其参数有一些变化，注意仔细观察。markerevery=[2, 4, 6]，即对第 2、4、6 个坐标点用规定的符号来标记（坐标点的计数依然从 0 开始计数，本例中第 0 个正好是坐标[0, 0]）。

In[7]和 In[8]的操作，是顺着前面“画函数曲线”的思路延续下来的，所以我们在参数中都有线型符号。如果思维跳跃一下，去掉表示线型的符号，图像会变成什么样？——这是学习的重要方法，要不断按照自己的理解来尝试，从而深化和拓宽对某个问题的认识。

```
In [9]: x = np.linspace(-np.pi, np.pi, 9)
        plt.plot(x,np.sin(x),'Dr',markersize=16,markerfacecolor='b',markevery=[2,4,6])
        plt.plot(x, np.cos(x), 'hr', markersize=16, markerfacecolor='m')
        plt.grid(True)
```

从图 3-3-7 中，能看出哪些点分别是正弦函数曲线和余弦函数曲线上的点吗？如果不看 In[9]，恐怕很难想到。

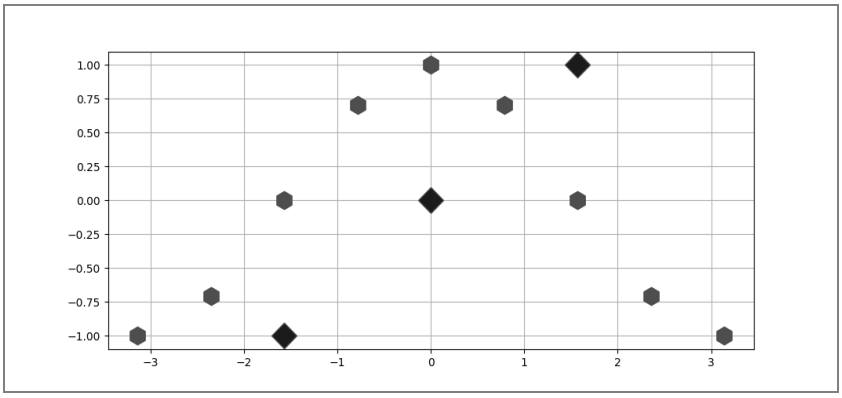


图 3-3-7　标记函数曲线上的点

有一种常用的科学方法，就是先通过试验，测出相关的量，然后根据所测的量在坐标系内描点，得到的可能就是类似图 3-3-7 那样的结果；再根据这个结果推测出那些点符合什么规律，并用函数表示该规律——在数据分析中，这个过程称为“回归”。当然，如果测量的数据有限，就很难找到正确的规律，于是就要多测一些数据。

数据量多了，规律就容易看出来了。当然，我们所举的示例都是比较简单的，在真实的研

究中，测量数据都有误差，也往往不是很简单的初等函数。

再回到“曲线”的属性，本质上就是 plot()函数参数，认清其本质，就知道实现方法了，那就是本书一再倡导的方法：“plt.plot()？”——请认真阅读文档。

线型种类有限，要区分不同的线，还可以通过其粗细，即 plot()函数中的 linewidth 参数。

```
In [10]: x = np.linspace(-np.pi, np.pi, 9)
         plt.plot(x, np.cos(x), linewidth=8)
```

曲线的平滑程度取决于坐标点的数量，In[10]中的 X 仅取了 9 个值，读者自行测试一下 X 取 100 个值的效果，并与图 3-3-8 所示的效果进行比较。

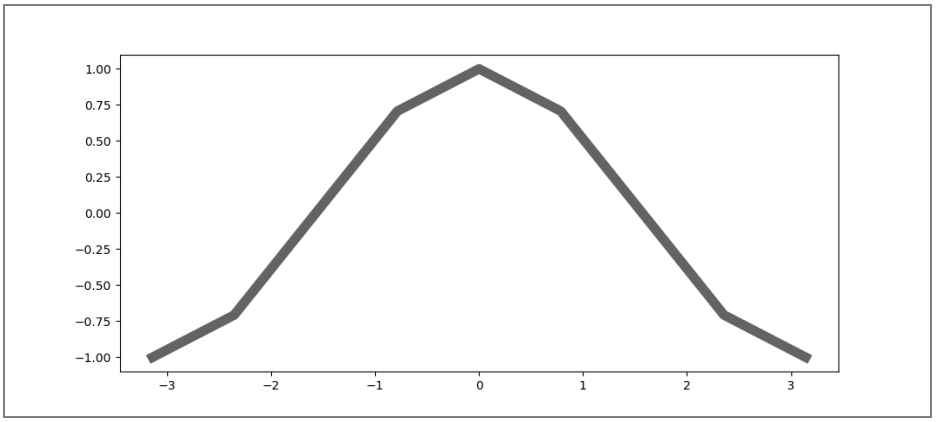


图 3-3-8　指定曲线的宽度

前面所画的图中，有的时候在一个坐标系内画出了多条曲线，比如图 3-3-4，虽然可以用线型和色彩对不同曲线进行区分，但是图中没有说明某个曲线代表的是什么含义。再看图 3-3-2，虽然线很多，但我们可以通过右上角的图例，对照着找到某条线的说明。

所以，图都要有图例，否则只是“画”。

2．图例

读者都看过地图——正规的地图，地图上都有 “图例”，用来说明地图中各个符号的含义等。与之类似，在 Matplotlib 所绘制的图中，如果也有图例，则会让图像更明确。

```
In [11]: a = np.arange(0, 3, .02)
         b = np.arange(0, 3, .02)
         c = np.exp(a)
         d = c[: : -1]
         plt.plot(a, c, 'k--', label="Model")
         plt.plot(a, d, 'r:', label="Data")
         plt.plot(a, c+d, 'b-', label="Total")
         plt.legend(loc='upper right')
```

在图 3-3-9 中绘制了三条曲线，为了注明每条线所表达的意义，使用了图例。

从程序中可以看到，实现对图例操作的函数是 plt.legend()，当然要理解它。笔者认为，对图例的各种操作中，最重要的就是位置。所以，下面重点阐述的也是图例在 Axes 容器中（坐标系中）的位置。

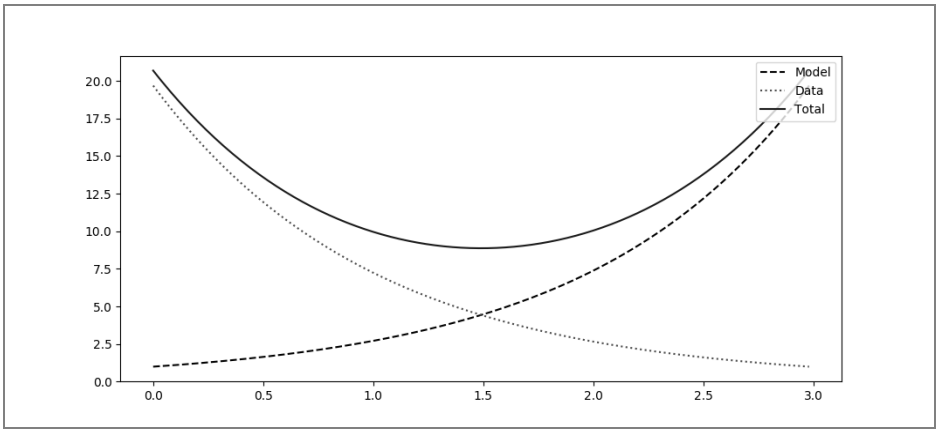


图 3-3-9　图例在右上角

在 plt.legend()函数中，通过参数 loc 来确定图例的位置，表 3-3-2 中就是这个参数可选的值，可以像 In[11]那样使用字符串（loc='upper right'），也可以使用编号，即 loc=1。

表 3-3-2　图例位置名称和编号

名　称	编　号	名　称	编　号
best	0	center left	6
upper right	1	center right	7
upper left	2	lower center	8
lower left	3	upper center	9
lower right	4	center	10
right	5		

```
In [12]: a = np.arange(0, 3, .02)
         b = np.arange(0, 3, .02)
         c = np.exp(a)
         d = c[: : -1]
         line1, = plt.plot(a, c, 'k--', label="Model")    #①
         line2 =plt.plot(a, d, 'r:', label="Data")[0]     #②
         line3, =plt.plot(a, c+d, 'b-', label="Total")
         plt.legend((line1, line2), loc=0)    #③
```

读者要注意 In[12]中的一些新技巧。语句①以赋值语句的方式通过变量 line1 获得相应的曲线对象，注意在 line1 后面有一个逗号，因为 plt.plot()返回的是一个列表，如果不按照①那样写，就得写成②那样。

再看语句③，与以往有所不同，最直接观察到的就是传入的参数多了，比 In[11]中多了(line1, line2)，其目的是告诉 legend()函数，产生的图例中只需要对 line1 和 line2 进行标识，不需要显示 line3。此参数也可以用 handles=(line1, line2)的方式设置。再就是 loc=0，这次没有使用图例位置名称字符串，而是使用了整数类型的编号。不过，读者要注意，loc=0 即是让 Matplotlib 选定一个最“best”的位置显示图例，Matplotlib 会根据图像窗口的情况调整图例显示位置。图 3-3-10 是笔者调试的时候所显示的，读者调试时可能有所不同。

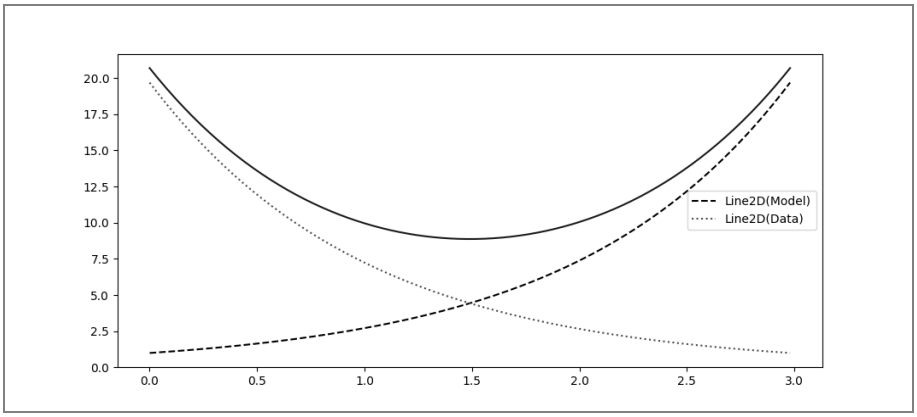


图 3-3-10　图例位置为 best

虽然表 3-3-2 中的位置已经不少了，但 Matplotlib 还不罢休，legend()函数中还有别的参数，允许我们更多样化地放置图例，从而满足多种自定义需求——在你的职业生涯中，一定会遇到很多“X 布斯”，不管真真假假，他们总要提出各种改变世界的伟大构想。如果 Matplotlib 中的图例不能自定义，“X 布斯”们就会很气愤，气愤之余就要自己造一个新的 Matplotlib——可是，他们还缺一个程序员，你！

```
In [13]: plt.plot([0,1,2,3,4,5], label="positive")
         plt.plot([5,4,3,2,1,0], label="negative")
         plt.legend(bbox_to_anchor=(0, 1), loc=3, ncol=2)
```

legend()函数的 bbox_to_anchor 参数能够将图例定位在任意位置，其值的形式是(x0, y0, width, height)，如图 3-3-11 所示。其实，bbox_to_anchor 参数的作用体现在两个方面：首先，它创建了一个“盒子”（box），上述元组形式的值本质上是这个“盒子”的位置和长宽；然后，图例被放置在这个“盒子”里面，图例在“盒子”中的相对位置用 loc 参数确定。

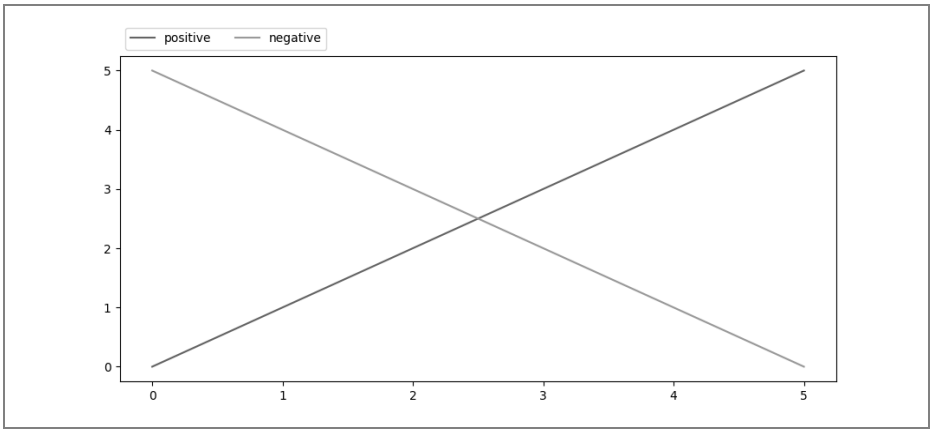


图 3-3-11　bbox_to_anchor 指定图例位置

- x0、y0：“盒子”的左下角相对于坐标左下角（原点）的位置，通常是 0~1 的浮点数，表示相对于 x、y 方向长度的比例。
- width、height：“盒子”的宽度和高度，其值也是 0~1 的浮点数，含义同上。

“盒子”位置指定之后，通过 loc 参数再确定图例在盒子中的相对位置，此处借用图 3-3-12 来说明 loc 和 bbox_to_anchor 参数的作用效果（图片来源：https://stackoverflow.com/questions/39803385/what-does-a-4-element-tuple-argument-for-bbox-to- anchor-mean-in-matplotlib/）。请读者注意观察、比较图中的 bbox_to_anchor 和 loc 参数值的特点及其显示结果。

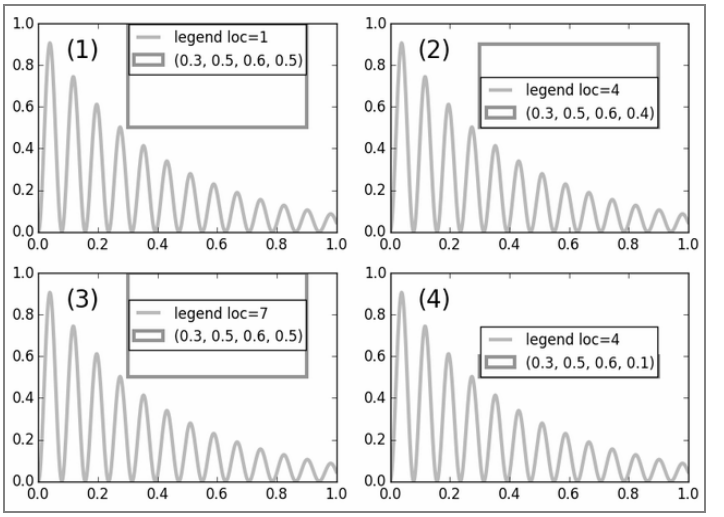


图 3-3-12　loc 和 bbox_to_anchor 参数效果

legend()的其他常用参数还有很多，读者可以通过查看文档来了解。下面通过示例，比较并体会几个常用参数带来的绘图效果，如图 3-3-13 所示。

```
In [14]: x = np.linspace(0, 10, 1000)
         fig, ax = plt.subplots(2, 2)    #创建 2×2 的子图对象

         #在每个分区的 Axes 对象中绘制曲线
         for i in range(2):
            for j in range(2):
                ax[i, j].axis('equal')
                line1, = ax[i, j].plot(x, np.sin(x), '-b', label='sin(x)')
                line2, = ax[i, j].plot(x, np.cos(x), '--r', label='cos(x)')

         #创建不同形式的图例
         ax[0, 0].legend(loc='upper right', frameon=False, fontsize=20)
         ax[0, 1].legend(loc='lower center', frameon=False, ncol=2, fontsize='large')
         ax[1, 0].legend(fancybox=True, framealpha=1, shadow=True,
                         borderpad=1,  fontsize='x-large')
         ax[1, 1].legend(handles=[line1], loc='upper right', fontsize='xx-large')
         #创建第一个图例

         from matplotlib.legend import Legend
         #创建 Legend 类实例对象
         leg = Legend(ax[1, 1], [line2], ['cos(x)'], loc='lower right',  fontsize=20)
         #等效于 plt.legend()
         ax[1, 1].add_artist(leg)    #增加第二个图例
```

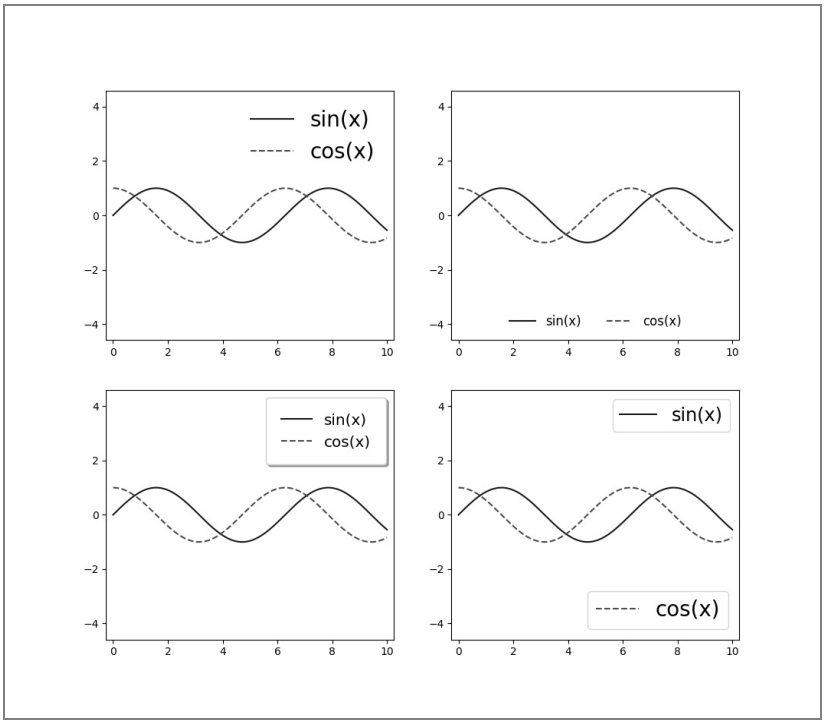


图 3-3-13　比较不同参数的图例

诚然，图例的样式还有很多种，本书在此处所讲述的仅仅是带领读者管窥而已，唯有通过阅读官方文档，并通过自己的调试，才能看到更多其他的样式。不过，不管是什么样式的图例，都要符合所绘之图的整体风格要求，并且不能喧宾夺主——图像才是主。

图像，除可以是曲线外，还可以有别的类型，比如“散点”。

3．散点图

前面所绘制的图都是根据已知的函数绘制的，还有另外一种情况，有一些横、纵坐标的数据，并且每个横坐标的数据都对应着唯一的纵坐标数据，但不知道它们之间的函数关系。这时候如果要在坐标系中绘图，只能绘制出一些分散的点，这种图就称为“散点图”。

```
In [15]: rng = np.random.RandomState(0)    #①
         x = rng.randn(100)    #②
         y = rng.randn(100)
         colors = rng.rand(100)    #③
         sizes = 1000 * rng.rand(100)
         plt.scatter(x, y, c=colors, s=sizes, alpha=0.3)    #④
         plt.colorbar()    #⑤
```

① 的 np.random.RandomState()是一个伪随机数发生器，通过此对象的方法获得随机数。

② 所得到的是一维数组，该数组的元素是 100 个符合高斯正态分布的随机数。

③ 得到的也是一个一维数组，它的元素是 100 个大于等于 0 且小于 1 的浮点数。

用于绘图的数据都是随机得到的，足够“散”，然后画图，这次不用 plt.plot()，而是用④中的 plt.scatter()，先看看它的执行结果，如图 3-3-14 所示。

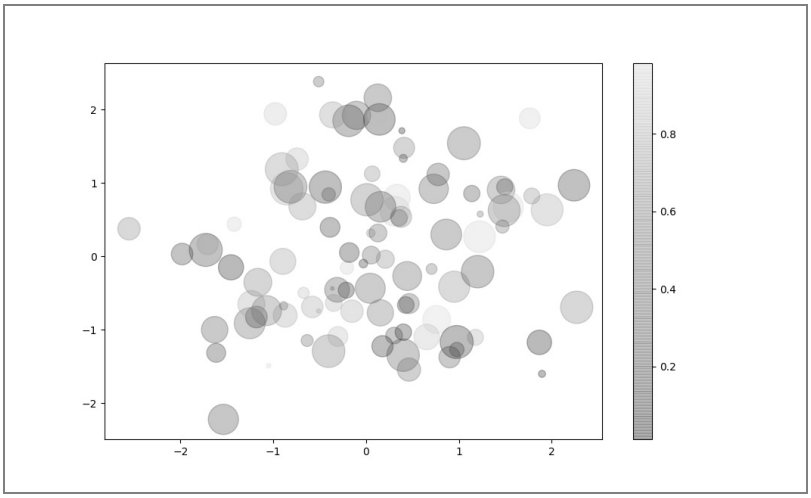


图 3-3-14　散点图

plt.scatter()中的参数 *x* 和 *y* 显然对应的是坐标系中的横、纵坐标，在本例中一共组合了 100 个点；c=colors 指定了每个点的颜色。跟 plt.plot()函数一样，plt.scatter()函数也有着众多的参数，请看下面完整的表述。

```
plt.scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, verts=None, edgecolors=None, hold=None, data=None, **kwargs)
```

读者要了解这些参数的含义，请自行查看文档，恕笔者在这里不做文档的翻译——尽管有的书如此。

⑤中又出现了一个新的函数，plt.colorbar()的作用是绘制图 3-3-14 右侧的彩色条——其名称的英文为 colorbar，有的资料就干脆把它叫作“彩色条”，但笔者觉得这种称呼比较乏味，由物理学受到的启发，笔者将其称为“数据光谱”。

画散点图的目的，除能够把数据比较直观地用图像表达出来外，更方便我们通过视觉观察这些数据的分布规律。

这其实就是一种简单的“回归”（regression）。

假设已经通过试验测量获得了一些数据（分别用 *x* 和 *y* 两个变量表示），试验测量中必然有误差，然后将这些数据在一个坐标系中描点，就得到了“散点图”，如图 3-3-15 所示。

观察图 3-3-15 中数据点的分布规律，可以猜测，*x* 和 *y* 之间应该具有线性关系，即图中数据所反映的规律应该是一个一元一次函数。考虑试验测量的误差因素，我们可以画一条直线，让图中的所有点比较均匀地分布在这条直线的两侧，那么这条直线所对应的函数，就是 *x* 和 *y* 之间的对应关系——这就是回归，如图 3-3-16 所示。

通过对散点图的观察，找到数据之间的函数关系，是散点图的一项应用。这项应用会在后面详述回归问题的时候继续深入研究。

“散点图”不仅可以用于探索数据规律，还可以直观地对一些数据进行描述。比如下面所展示的综合应用中，就是以“散点图”的方式展示江苏省各个城市的面积人口和所在位置（数据文件 city_population.csv 可以到本书代码仓库中下载）。

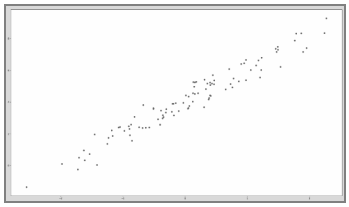


图 3-3-15　寻找散点图的规律

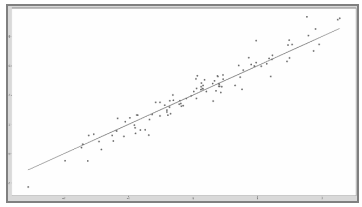


图 3-3-16　线性回归

```
In [16]: cities = pd.read_csv(" /home/qiwsir/Documents
                              /DataAnalysis/chapter03/city_population.csv")
         lat = cities['latd']
         lon = cities['longd']
         population = cities['population'],
         area = cities['area']

         plt.scatter(lon, lat, label=None, c=np.log10(population),
                     cmap='viridis', s=area, linewidth=0, alpha=0.5)
         plt.axis(aspect='equal')
         plt.xlabel("longitude")
         plt.ylabel('latitude')
         plt.colorbar(label='log$_{10}$(population)')    #①

         for area in [100, 300, 500]:
             plt.scatter([], [], c='k', alpha=0.3, s=area, label=str(area) + ' km$^2$')

         plt.legend(scatterpoints=1, frameon=False, labelspacing=1, title='City Area')

         plt.title('Jiangsu Cities: Area and Population')    #②
```

city_population.csv 是保存了江苏省各个城市相关信息的文件，其内容如图 3-3-17 所示（注意：不保证表中数据准确无误，在这里仅仅是为了演练上述代码而已）。

cities

	name	area	population	longd	latd
0	南京市	6582.31	8004680	118.78	32.04
1	无锡市	4787.61	6372624	120.29	31.59
2	徐州市	11764.88	8580500	117.20	34.26
3	常州市	4384.57	4591972	119.95	31.79
4	苏州市	8488.42	10465994	120.62	31.32
5	南通市	8001.00	7282835	120.86	32.01
6	连云港市	7615.29	4393914	119.16	34.59
7	淮安市	9949.97	4799889	119.15	33.50
8	盐城市	16972.42	7260240	120.13	33.38
9	扬州市	6591.21	4459760	119.42	32.39
10	镇江市	3840.32	3113384	119.44	32.20
11	泰州市	5787.26	4618558	119.90	32.49
12	宿迁市	8555.00	4715553	118.30	33.96

图 3-3-17　江苏省各个城市的相关数据（仅为了演示，因此数据可能与实际不同）

语句①实现数据光谱（colorbar），因为显示的是人口数量，往往数值较大，所以在显示的时候进行了数学处理——取对数，并通过参数 label='log$_{10}$(population)'在数据光谱旁边标明，注意下标的写法，在此句的下面还有上标的写法。这种写在“$”符号之间的标记，被称为 TeX 标记——关于 TeX，推荐阅读 https://zh.wikipedia.org/wiki/TeX——这是一个“大神”的作品，在此献上笔者的敬意。

语句②的作用是在坐标系（Axes 对象）上面为该图增加一个标题。

In[16]的代码不仅仅是要画出图 3-3-18，更是带领读者将已学的知识进行综合运用，再适当增加一点新知识。所以，特别建议暂时停止向下阅读，而是返回到 In[16]，对每行代码进行注释，从而理解其含义。如果在阅读完所有代码之后，闭上眼睛，脑海中浮现了相应的图像，则实现了“人码合一”。

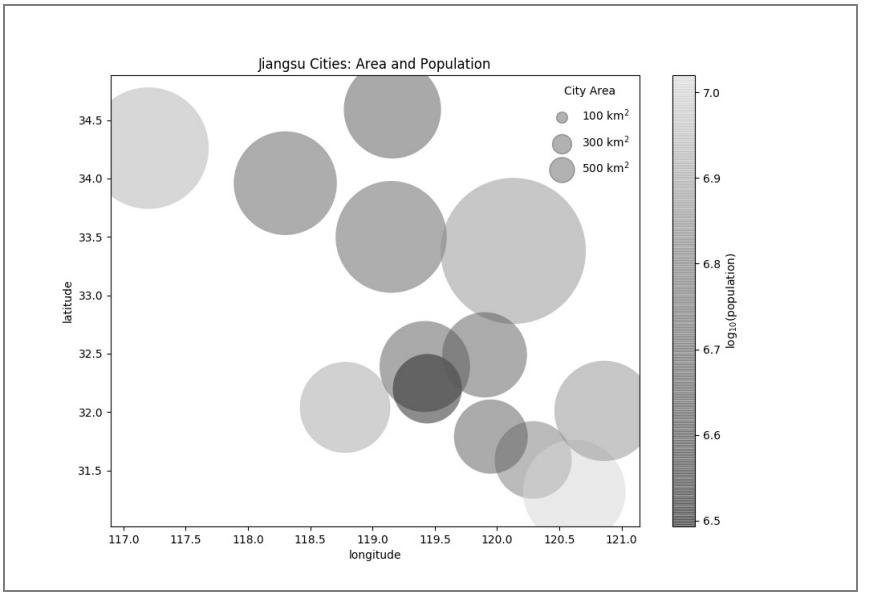


图 3-3-18　江苏省各城市人口和面积经纬度散点图

特别建议读者打开江苏省行政图，跟图 3-3-18 比较一下，从而可以更清晰地理解 In[16]代码所要达成的目标。

请返回到 In[8]的代码，在那里我们通过“plt.plot(x,np.sin(x),'Dr',markersize=16, markerfacecolor='b', markevery=[2,4,6])”也在坐标系中画了一些“点”。通过 plot()画出的“点”与通过 scatter()画出的点有何不同？

从上面的作图过程中不难发现，scatter()会根据点的大小、颜色对每个点进行渲染，这样各个点是不相同的，比如图 3-3-18 中各个城市的人口和面积不同，那么点的色彩和半径也不同；而 plot()中的点都是相同的，由相关参数统一确定其样式和色彩。所以，两者就有了不同的用途。

scatter()适合绘制数据量较小的散点图，当数据量较大时，它所耗费的时间就会相应变长了。

至此，读者已经了解了 Matplotlib 绘图的基本方法。在实际应用中，它还提供很多常用的绘图函数，比如常见的统计图。

3.4　常用统计图

虽然使用 plot()函数能够画出各种函数曲线，用 scatter()能够画出散点图，但常用的统计图还有很多别的样式。比如在微软的 Excel 中，就提供了绘制常用统计图的不少模板，如图 3-4-1 所示，如果读者所处理的数据不是很大，不妨使用 Excel——选用什么工具要因地制宜，不要拿着锤子看任何东西都是钉子。

图 3-4-1　Excel 提供的常用统计图模板

与之相比，Matplotlib 的优势在于面对大量数据的时候，使用它所定义的函数就如同用 Excel 绘制统计图那样便捷，只不过 Matplotlib 是用代码实现的。用代码实现，也就意味着定制性更强。

所以，在面对大数据、复杂多变的业务时，必须用代码解决，这是“码农”的价值所在。

能解决复杂问题的，“身价”自然就高了。

但解决复杂问题的工具，也要趁手才行。

Matplotlib 满足此要求，它有成熟的绘制统计图方法——除 plot()和 scatter()外。

1．柱形图

柱形图是一种常见的统计图，它也有很多具体形态，从图 3-4-1 所示的 Excel 的柱形图模板中就能看出来。在 Matplotlib 中绘制柱形图的函数是 bar()，它通过参数来确定图的形态。

```
plt.bar(left, height, width=0.8, bottom=None, hold=None, data=None, **kwargs)
```

下面通过示例讲解各参数的具体用途。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
In [2]: data = [2, 10, 4, 8, 6]
        position = [1, 2, 3, 4, 5]
        plt.bar(left=position, height=data)
```

柱形图中“柱子”的基本位置和长、宽，是由 left、height、width、bottom 确定的。

- left：默认情况下，其值是“柱子”竖直中线的位置，从图 3-4-2 中可见，每个“柱子”的竖直中线依次在 position 所示的位置。
- height：“柱子”的高度，也可以将 left=position 理解为 *X* 轴的数据，height 则为相应的 *Y* 轴的数据。
- width=0.8：“柱子”的宽度，默认是 0.8。
- bottom：“柱子”底部与 *X* 轴的距离，默认为 None，即距离都为 0。

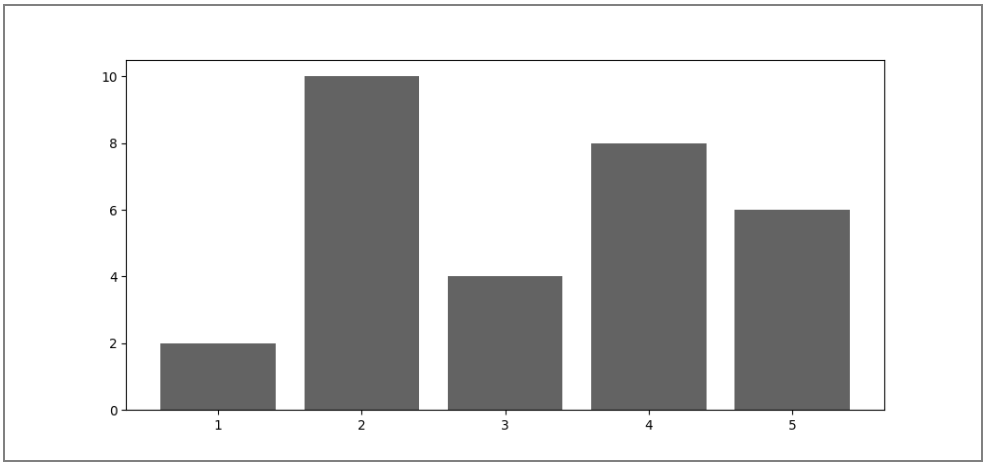


图 3-4-2　基本柱形图

```
In [3]: data = [2, 10, 4, 8, 6]
        position = [1, 2.5, 3, 4.5, 5]
        plt.bar(left=position, height=data, width=0.4, bottom=[3, 0, 5, 0, 1])
        plt.grid(True)
```

修改参数后，柱形图如图 3-4-3 所示。

比较 In[3]和 In[2]中 bar()函数的参数，以及生成的两张图，前面所阐述的四个基本参数的作用就更明朗了。

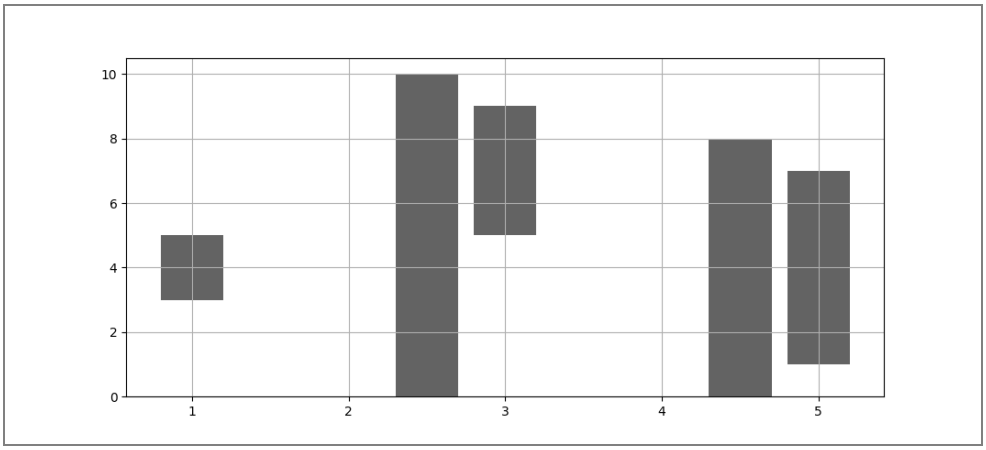


图 3-4-3　修改参数后的柱形图

不仅如此，还能装饰“柱子”，让它多彩。

```
In [4]: data = [2, 10, 4, 8, 6]
        position = [1, 2, 3, 4, 5]
        labels = ['Pejing', 'Soochow', 'Hongkang', 'Tsingtao', 'Canton']
        plt.bar(left=position, height=data, width=0.4,
                color='gb', edgecolor='r', linestyle='--',
                linewidth=3, hatch='x', tick_label=labels)
```

这次增加了好几个新参数，它们的作用在于装饰“柱子”，如图 3-4-4 所示。

- color='gb'：设置“柱子”的颜色，这里规定了两种颜色，g 表示 green，b 表示 blue，如果只规定一种，比如 color='g'，则所有“柱子”颜色都是绿色。
- edgecolor='r'：设置“柱子”边缘的颜色。
- linestyle='--'：设置“柱子”边线的线型。
- linewidth=3：设置“柱子”边线的宽度。
- hatch='x'：填充“柱子”内部的图形。
- tick_label=labels：设置 *X* 轴刻度的标示，替代默认的数字标示。

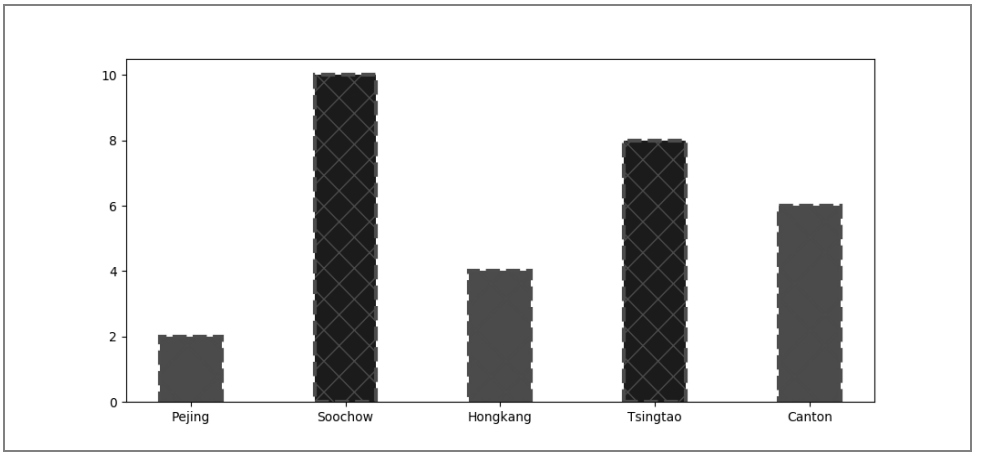


图 3-4-4　多彩的柱形图

尽管柱形图已经五彩缤纷了，但以上还属于基本的操作。

言外之意，还有高级的。

```
In [5]: position = range(1, 6)
        a = np.random.random(5)
        b = np.random.random(5)

        plt.bar(position, a, label='a', color='b')    #①
        plt.bar(position, b, bottom=a, label='b', color='r')    #②

        plt.legend(loc=0)
```

这里使用了两次 bar()函数，①与②的区别在于 bottom 参数，②中规定“柱子”与 *X* 轴之间的距离是 *A*，也就正好出现如图 3-4-5 所示的效果了——堆积柱形图。

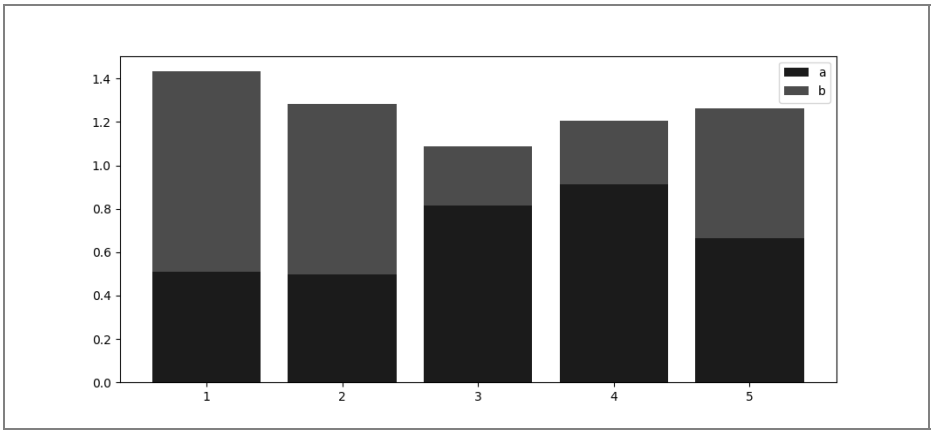


图 3-4-5　堆积柱形图

除堆积柱形图外，还有一种“簇状柱形图”。所谓“簇状”就是几个“柱子”一组，同一组的“柱子”之间的距离应该是比较小的，通常最小就是 0，如图 3-4-6 所示。

```
In [6]: position = np.arange(1, 6)
        a = np.random.random(5)
        b = np.random.random(5)

        total_width = 0.8    #①
        n = 2
        width = total_width / n
        position = position - (total_width - width) / n    #②

        plt.bar(position, a, width=width, label='a', color='b')    #③
        plt.bar(position + width, b, width=width, label='b', color='r')    #④

        plt.legend(loc=0)
```

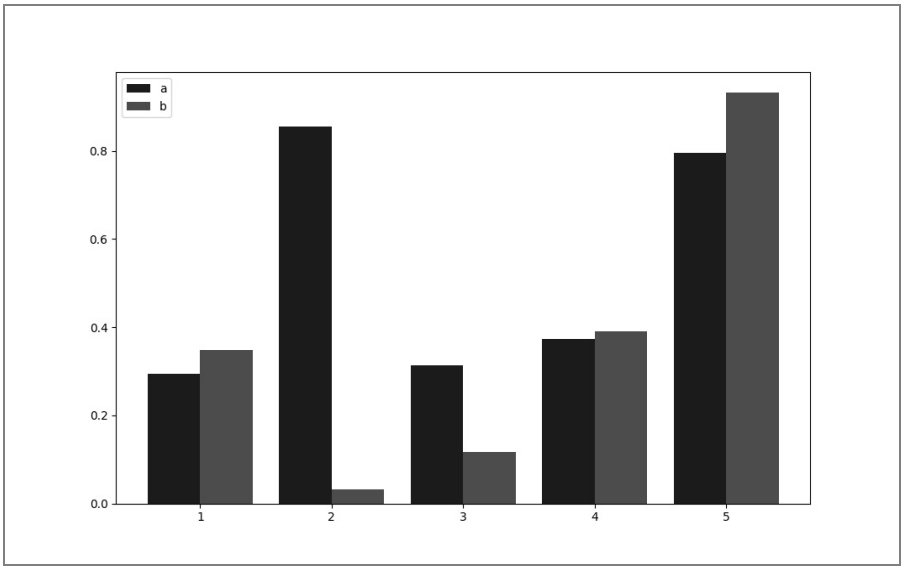


图 3-4-6　簇状柱形图

In[6]中的语句①和②重新规定了每簇的位置。③绘制第一个“柱子”，因为每簇的“柱子”之间距离为 0，所以在④中以 position+width 作为第二个柱子的中心线位置。

2．条形图

按照 Excel 中的习惯说法，将“柱子”横过来就是“条”，那么所谓条形图的形状也能想象出来了。

```
In [7]: position = np.arange(1, 6)
        a = np.random.random(5)

        plt.barh(position, a)
```

绘制条形图，所使用的函数是 barh()，函数的名称就告诉我们，条形图是由柱形图（bar）转为水平（horizontal）而来的，所以，其参数所指也随之顺时针转 90°，如图 3-4-7 所示。按照这种思路，读者就可以绘制多种形态的条形图了，包括“堆积”“簇状”。

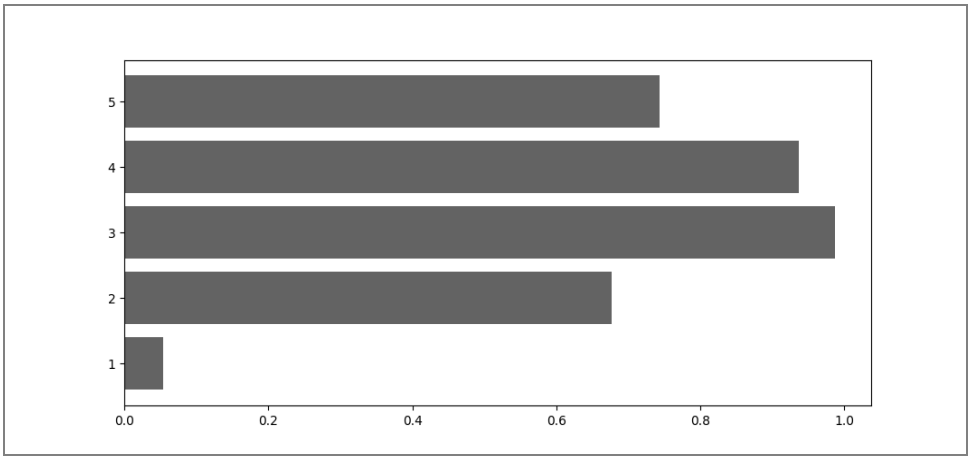


图 3-4-7　条形图

在条形图中，有一种称之为“正负条形图”的，或许在比较某些对象的时候会用到，如图 3-4-8 所示。

```
In [8]: position = np.arange(1, 6)
        a = np.random.random(5)
        b = np.random.random(5)

        plt.barh(position, a, color='g', label='a')
        plt.barh(position, -b, color='r', label='b')

        plt.legend(loc=0)
```

条形图函数 plt.barh()的使用方法与 plt.bar()类似，所以此处不再赘述，读者可以进一步阅读其相关文档。

3．箱线图

英语中的“Box plot”有多种翻译，如盒须图、盒式图、盒状图、箱线图、箱形图等，不管什么名称，它的基本结构如图 3-4-9 所示。

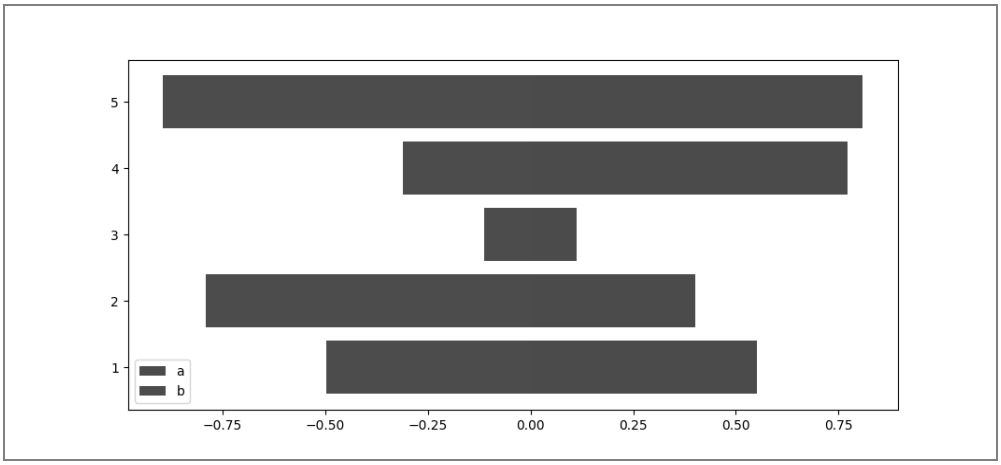


图 3-4-8　正负条形图

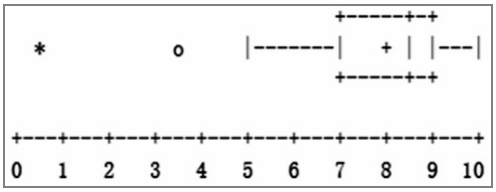


图 3-4-9　箱线图示例（图示来自《维基百科》“箱形图”词条）

这种图是美国著名统计学家约翰 • 图基（John Tukey）于 1977 年发明的。它能显示出一组数据的最大值、最小值、中位数及上下四分位数。以图 3-4-9 为例，我们依次可知：

- 最小值(minimum)=5；
- 下四分位数(Q1)=7；
- 中位数(Med，也就是 Q2)=8.5；
- 上四分位数(Q3)=9；
- 最大值(maximum)=10；
- 平均值=8；
- 四分位间距(interquartile range)= Q3-Q1=2 (即ΔQ)。

除直观阅读到数值外，还可以进一步计算其他的数值，此方面的知识请参阅统计学的相关资料。

我们在这里要研究的是怎么用 Matplotlib 画出这种图，如图 3-4-10 所示。

当然要有一个函数——boxplot()——从简单的开始。

```
In [9]: fig,ax = plt.subplots(1, 2)
        data = [1, 5, 9, 2]
        ax[0].boxplot([data])    #①
        ax[0].grid(True)
        ax[1].boxplot([data], showmeans=True)    #②
        ax[1].grid(True)
```

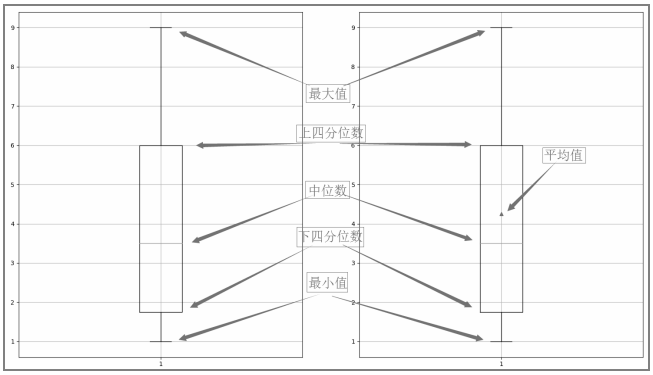


图 3-4-10　简单的箱线图

①是最简单的使用方式，默认没有显示平均值，②通过 showmeans=True 将平均值显示出来。

In[9]中使用的是 Axes 对象的 boxplot()方法绘制箱线图，当然也可以使用 plt.boxplot()绘制。此外，Pandas 的数据对象也具有 boxplot()方法。

```
In [10]: np.random.seed(12345)
         data = pd.DataFrame(np.random.rand(5,4), columns=["A", 'B', 'C', 'D'])
         bp = data.boxplot(sym='r*', vert=False, meanline=False, showmeans=True)
```

使用 In[10]的代码，得到了水平放置的箱线图，如图 3-4-11 所示。其中起作用的参数是 vert=False，如果 vert=True，则为竖直方向（默认竖直）。另外几个参数的含义如下。

- sym='r*'，表示异常点的形状。
- meanline=False，表示平均值的形状不能为线性，因为中位数常常用线段表示，这样做可避免两者被混淆。

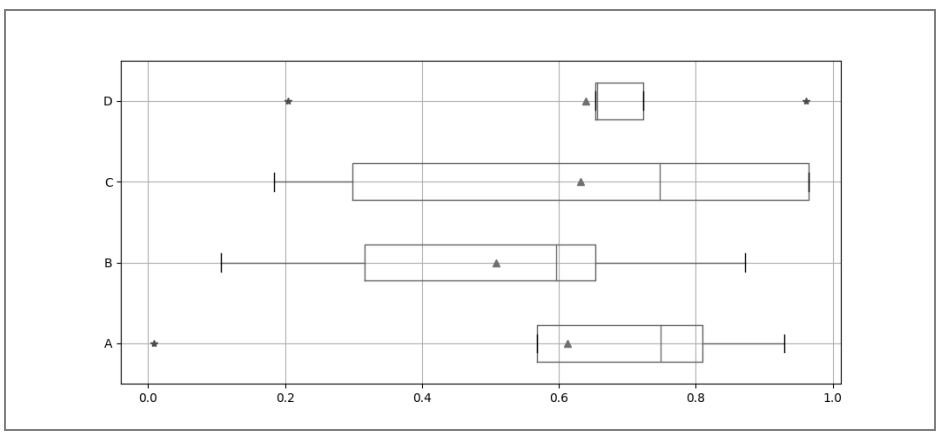


图 3-4-11　水平箱线图

如果读者通过“plt.boxplot?”方法查看文档，会看到还有很多其他的参数，当然，有些参数很少被使用，读者也只需要了解一二，以备将来用。

对于箱线图而言，除使用参数外，还可以用其他方式进行装饰。

```
In [11]: np.random.seed(123)
         d1 = np.random.normal(100, 10, 200)
```

```
d2 = np.random.normal(80, 30, 200)
d3 = np.random.normal(90, 20, 200)
d4 = np.random.normal(70, 25, 200)
data = [d1, d2, d3, d4]

fig = plt.figure(1, figsize=(9, 6))
ax = fig.add_subplot(111)
bp = ax.boxplot(data, patch_artist=True)

for box in bp['boxes']:
    box.set(color='#666600', linewidth=2)
    box.set(facecolor='#CCCCCC')

for whisker in bp['whiskers']:
    whisker.set(color='#009933', linewidth=6)

for cap in bp['caps']:
    cap.set(color='#660066', linewidth=2)

for median in bp['medians']:
    median.set(color='#663300', linewidth=2)

for flier in bp['fliers']:
    flier.set(marker='^', color='#990033', alpha=0.5)
```

从图 3-4-12 的效果中可以看到，我们使用了各个对象的 set()方法，将箱线图装饰了一番。当然这里不是为了美化，纯粹是为了显示各个参数效果。如何理解 In[11]的代码呢？建议读者使用“控制变量”法对 In[11]中的每个 set()方法进行调试，从而理解代码的含义——不用笔者讲述，读者一定能理解，并且一定要努力去理解。

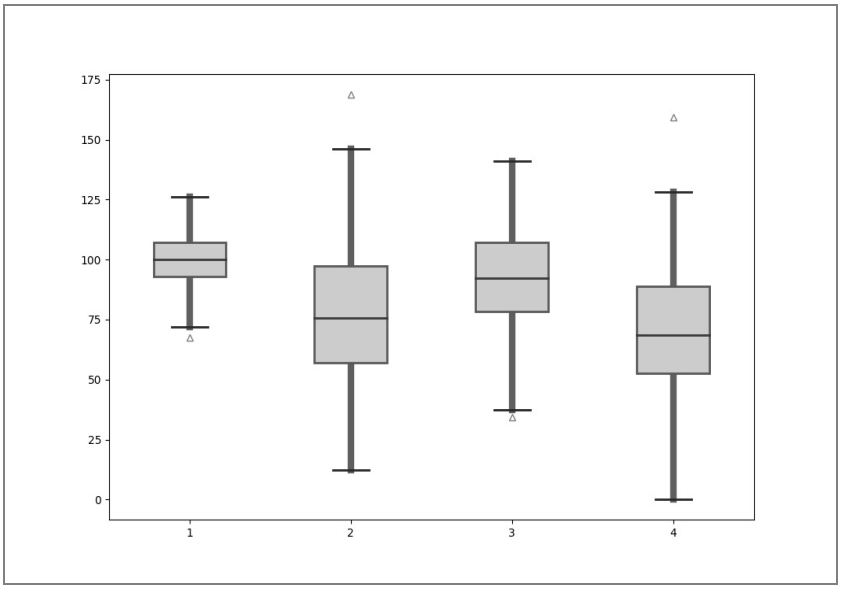


图 3-4-12　装饰之后的箱线图

4．饼图

饼图也是常用的统计图表，它显示一个数据系列中各项的大小与各项总和的比例。在 Matplotlib 中，使用 pie()方法绘制饼图，其完整形式是：

```
plt.pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistance=0.6,
shadow=False, labeldistance=1.1, startangle=None, radius=None, counterclock=True,
wedgeprops=None, textprops=None, center=(0, 0), frame=False, hold=None, data=None)
```

老调重弹，依然推荐读者现在或者学习完本节之后，阅读一下“plt.pie?”文档的内容。

```
In [12]: x = [2, 4, 6 ,8]
         fig, ax = plt.subplots()
         labels = ['A', 'B', 'C', 'D']
         colors = ['red', 'yellow', 'blue', 'green']
         explode = (0, 0.1, 0, 0)
         ax.pie(x, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%',
                shadow=True, startangle=90, radius=1.2)    #①
         ax.set(aspect="equal", title='Pie')
```

利用 In[12]的代码，画出了一张漂亮的饼图，如图 3-4-13 所示。跟以往的图形一样，参数是控制饼图形状的关键，这里对①中的参数做简要说明。

- x：数据源。
- explode：“扇面”的偏离。图 3-4-13 是一个“饼”，被分成了 4 个“扇面”，explode 中第二个数是 0.1，对应 B“扇面”偏离 0.1，其他数为零，即不偏离。
- labels：为每个“扇面”设置标示。
- colors：为每个“扇面”设置颜色。
- autopct：按照规定格式在每个“扇面”上显示百分比。
- shadow：是否有阴影。
- startangle：第一个“扇形”开始的角度，然后默认依逆时针旋转。
- radius：半径大小。

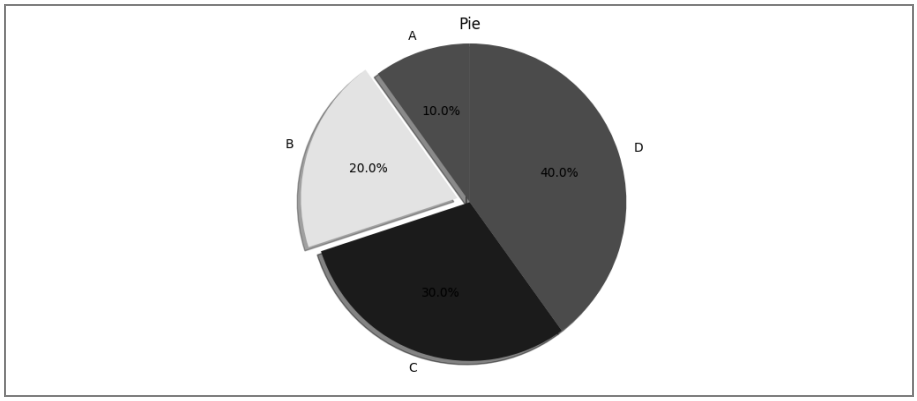


图 3-4-13　饼状图

仅看代码示例了解参数的含义，还不算掌握了用法。“知道”不等于“技能”，关键在于多练习。所以，读者要在阅读了饼图基本画法之后，再练习一番。

5．直方图

前面绘制了“柱形图”和“条形图”，这两者有很多相似的地方，仅有“竖直”和“水平”之别。但此处的“直方图”，与以上两种图形相比，差别可以用大相径庭来形容。

为了说明什么是直方图，笔者在这里借用《概率论与数理统计》（盛骤等编著，浙江大学出版社，2008 年 6 月第 4 版第 1 次印刷）里面的一个例题（题目的解会根据本书语言风格需要做适当修改），通过这个例题理解“直方图”含义，以及其与“柱形图”的区别。

题目：下面列出了 84 个伊特拉斯坎（Etruscan）人男子的头颅的最大宽度（mm），现在来画这些数据的“频率直方图”。

141, 148, 132, 138, 154, 142, 150, 146, 155, 158, 150, 140, 147, 148, 144, 150, 149, 145, 149, 158, 143, 141, 144, 144, 126, 140, 144, 142, 141, 140, 145, 135, 147, 146, 141, 136, 140, 146, 142, 137, 148, 154, 137, 139, 143, 140, 131, 143, 141, 149, 148, 135, 148, 152, 143, 144, 141, 143, 147, 146, 150, 132, 142, 142, 143, 153, 149, 146, 149, 138, 142, 149, 142, 137, 134, 144, 146, 147, 140, 142, 140, 137, 152, 145

解：以上数据最小值、最大值分别为 126、158，所有数据都落在区间[126, 158]上。那么，如果设定一个区间[124.5, 159.5]，则所有的数据也在此区间内——没有什么理由，区间大点，略有冗余，这是画直方图的通常做法。

将区间[124.5, 159.5]划分为 7 小段（原书称为“小区间”），每小段的长度用希腊字母Δ表示，则Δ=（159.5 – 124.5）/ 7 = 5。给Δ一个文雅的名称，叫作“组距”，即将区间[124.5, 159.5]划分为 7 个组，每组的组距为 5（如图 3-4-14 所示），每组的两端（如 124.5、129.5）称为“组限”。

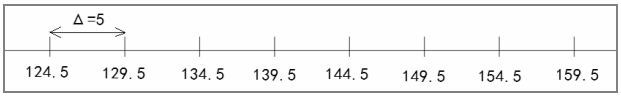


图 3-4-14　组距图示

数一数落在各个组内的数据个数——严格的概念叫“频数”，用 f 表示。各组的频数与总数据量（84）的比值称为频率，即 f/n，如表 3-4-1 所示。

表 3-4-1　直方图频率表

组　　限	频数 f	频率 f / n	累积频率
124.5 ~ 129.5	1	0.0119	0.0119
129.5 ~ 134.5	4	0.0476	0.0595
134.5 ~ 139.5	10	0.1191	0.1786
139.5 ~ 144.5	33	0.3929	0.5715
144.5 ~ 149.5	24	0.2857	0.8572
149.5 ~ 154.5	9	0.1071	0.9524
154.5 ~ 159.5	3	0.0357	1

若以图 3-4-14 的每个小段（如 124.5 到 129.5 之间的线段）为宽，以频数 f 为高，绘制小矩形，得到的就是“频数直方图”，如图 3-4-15 所示。若仍以每个小段为宽，以 $f/(n\Delta)$的值为高，画出一系列的小矩形，得到就是“频率直方图”，如图 3-4-16 所示。

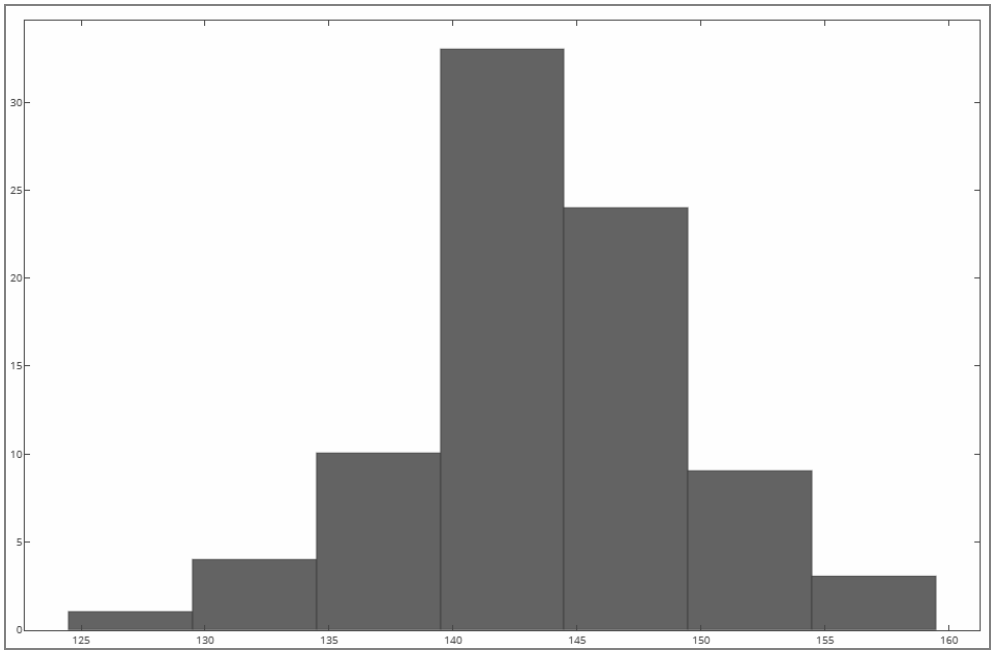


图 3-4-15　频数直方图（注意观察纵坐标）

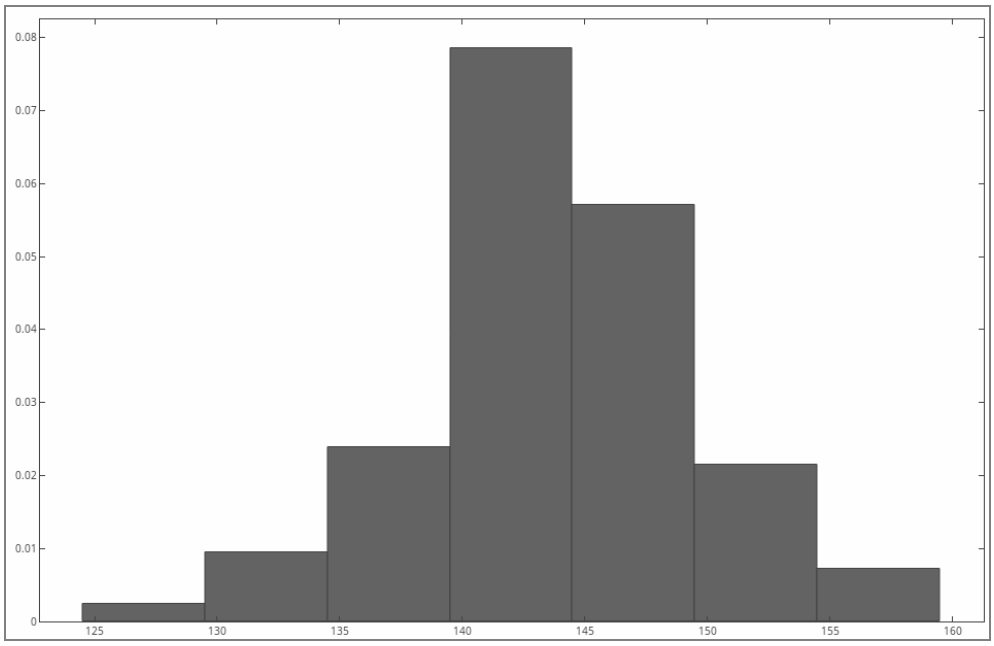


图 3-4-16　频率直方图（注意观察纵坐标）

有了上述储备知识，就可以学习下面的代码了。

稍等，在执行代码之前，要向读者介绍一个很好的数据可视化网站——https://plot.ly/。为什么要在这里介绍？一是因为下面要用到，二是对于那些有毅力坚持阅读到这里的读者，当然要为其提供一些好东西。

第一步，要在网站（https://plot.ly）注册一个自己的账户。

第二步，登录网站，到 Settings 里面找到自己的 API Keys，或者直接打开 https://plot.ly/settings/api，如图 3-4-17 所示。

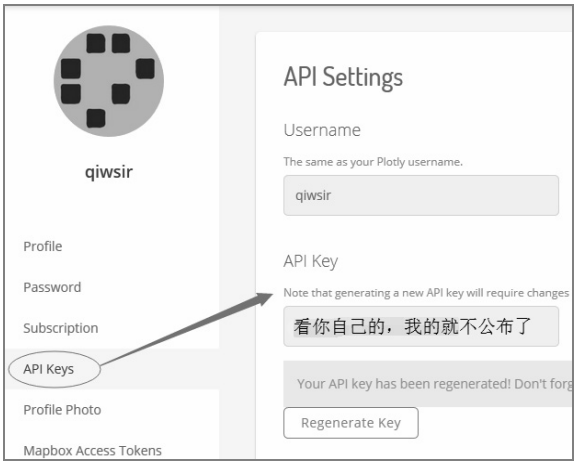


图 3-4-17　plot.ly 的个人账号 API Keys

第三步，按照 https://plot.ly/python/getting-started/文档（如果访问困难，请上网搜索安装方法）的说明安装第三方库。

```
$ sudo pip install plotly
```

以上准备工作完毕，即可在 Jupyter 中使用了。

```
In [13]: import plotly
         plotly.tools.set_credentials_file(username="用户名",api_key="本用户的 API Keys")
```

执行 In[13]的程序，没有任何反应，就对了。

```
In [14]: import plotly.plotly as py
         datas = np.random.randn(1000)
         plt.hist(datas)     #①
         fig = plt.gcf()    #②
         plot_url = py.plot_mpl(fig, filename="laoqi-basic-hist")    #③
```

执行 In[14]代码，会打开一个网页（本例中打开 https://plot.ly/%7Eqiwsir/0/），在本地不再显示图示。笔者调试 In[14]代码时得到的页面如图 3-4-18 所示，当读者调试的时候，可能会由于网站方对页面的修改而有所不同。

如此，一张惊艳的统计图就可以通过 URL 向外分享了，而不必像以往那样，只保存在自己的计算机中。

Plotly 就是数据可视化领域的 GitHub。

体会了成功的喜悦之后，还是平复心情，看看 In[14]中几个语句的含义吧。

① 执行 plt.hist()方法，绘制直方图（严格地说是“频数直方图”）。这个方法的参数不少，不过相信读者已经有了经验，只要阅读文档，一定能搞清楚。

② 的作用是得到当前所绘制的图对象，plt.gcf()的意思是“Get the Current Figure”。

③ 就是将 fig 对象发送到 plot.ly 上，并返回其地址。

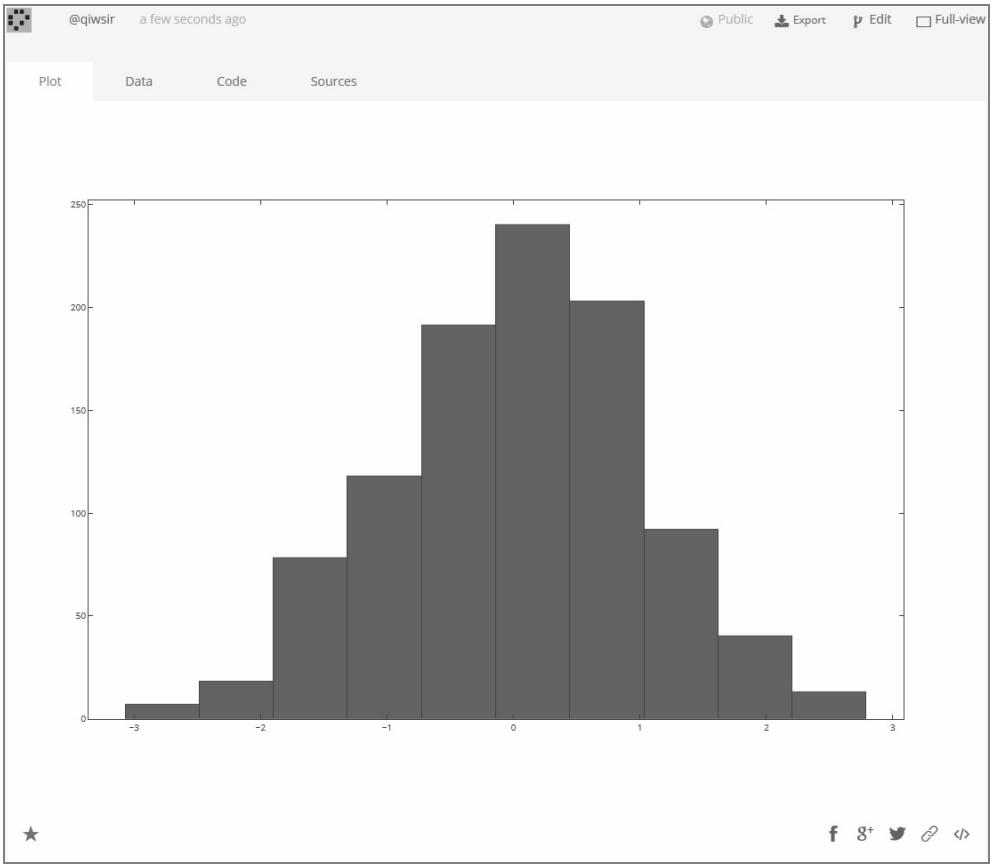


图 3-4-18　简单的直方图

现在我们对比一下“直方图”和“柱形图”。

绘制柱形图时，每个“柱子”单独渲染；直方图则不必如此，所以，它可以针对海量数据绘图——颇似 scatter()与 plot()的关系。

再从统计学角度来看，柱形图中用“柱子”长度表示各类别频数的多少，“柱子”的宽度则是根据需要而定义的值；直方图中每个矩形的高度表示该组的频数或 $f/(n\Delta)$，宽度则表示组距。

继续看代码，并深入理解。

```
In [15]: import matplotlib.mlab as mlab
         fig = plt.figure()
         mu = 100     #平均值
         sigma = 15     #标准差
         x = mu + sigma * np.random.randn(10000)    #①

         num_bins = 50    #②
         n, bins, patches = plt.hist(x, num_bins, normed=True, facecolor='blue', alpha=0.5)
         #③
         #绘制正态分布曲线
         y = mlab.normpdf(bins, mu, sigma)    #④
         plt.plot(bins, y, 'r--')
```

```
plot_url = py.plot_mpl(fig, filename='laoqi-mu-hist')
```

打开 https://plot.ly/~qiwsir/2/-line0/，看到了一张美轮美奂的图像，这次是“频率直方图”。当然，读者可以修改 In[15]中的某些参数，比如 num_bins=100，绘制出来的图像会更加细腻，如图 3-4-19 所示。

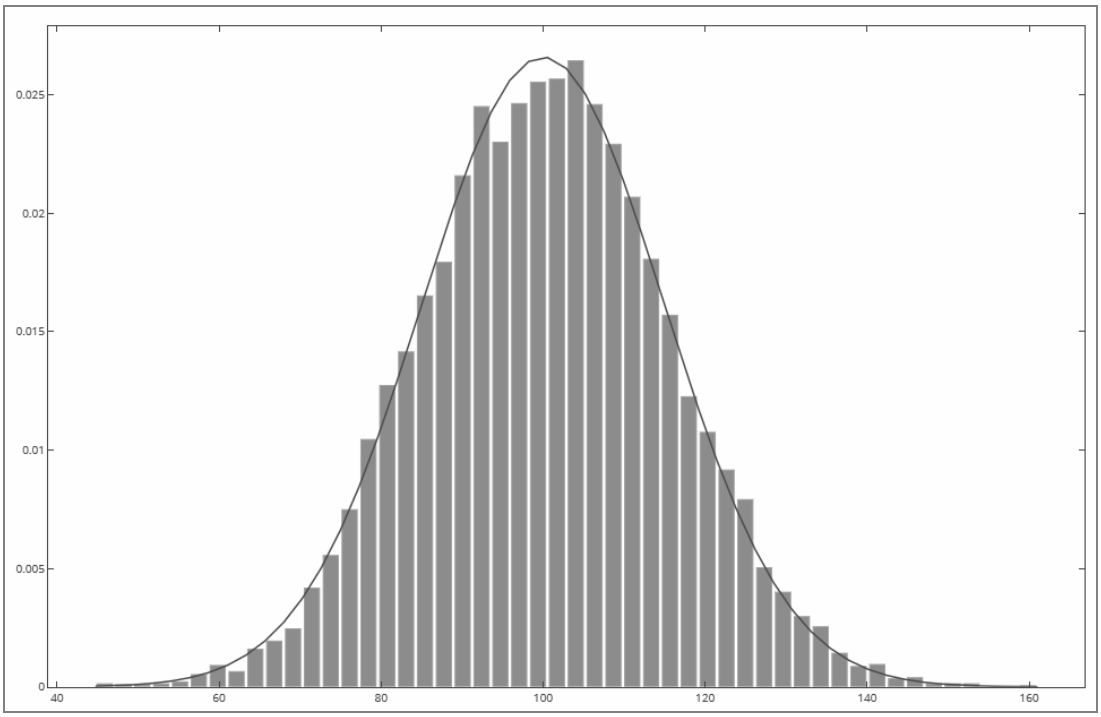


图 3-4-19　正态分布与直方图

为了让思路更加明朗，必须对 In[15]中的某些语句做一些解释。

①得到了 10000 个整数。如果对这些整数进行计数，则它们的分布符合正态分布规律。并且，这些整数的平均值是 100，标准差是 15。

②将 10000 个整数所在区间分为 50 个组（小区间、小段）。

③是关键，先看其完整的参数列表，这里仅介绍两个重要参数。

```
plt.hist(x, bins=None, range=None, normed=False, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, hold=None, data=None, **kwargs)
```

- bins：如果是整数，则表示分组的数量，默认是 10。在③中 bins=num_bins，表示此处划分为 50 个小组。如果是类列表的序列，比如[40, 80, 120, 160, 200]，则表示划分为[40, 80)、[80, 120)、[120, 160)、[160, 200]这样的 4 组。当然，这里也可以规定组距不均匀分布。
- normed：若为 True 或 1，则绘制“频率直方图”（直方图面积总和为 1——归一化）；默认为 False，绘制“频数直方图”。

③中还有三个返回值，n 为频数列表；bins 为各组的“组限”值列表；patches 则为用于创建直方图的数据集。

当然，plt.hist()中还有很多别的参数，请读者阅读文档并尽可能尝试。

④得到的是概率密度，即

$$f(x)=\frac{1}{\sigma\sqrt{2\pi}}\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

至此，我们学习了 5 种统计图，其实 Matplotlib 所提供的绘图函数还有很多，比如：

- 绘制平行于 X 轴的直线——plt.axhline()函数；
- 绘制平行于 Y 轴的直线——plt.axvline()函数；
- 绘制平行于 X 轴的区间带——plt.axhspan()函数；
- 绘制平行于 Y 轴的区间带——plt.axvspan()函数；
- 绘制梯状图——plt.step()函数；
- 绘制小提琴图——plt.violinplot()函数；
- 绘制误差图——plt.errorbar()函数；
- 绘制填充图——plt.fill_between()函数；

……

一言难尽啊，太多了。

不用惊慌，如果读者已经掌握了本书所倡导的学习方法，那么遇到任何一个新的问题，都能泰然处之，最终搞定。

3.5　绘制三维图像

除能在二维坐标系中实现数据可视化外，还能在三维坐标系中绘制三维图像。本节将简要介绍如何使用 Matplotlib 实现三维图像的绘制。

在 Matplotlib 中有一个专门绘制三维图像的工具 mplot3d，当 Matplotlib 安装之后，它也被同时安装在本地了。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from mpl_toolkits import mplot3d
```

In[1]中的 from mpl_toolkits import mplot3d 引入了 mplot3d 子模块，有了它之后，就可以如同 In[2]所示那样，在坐标轴的方法中使用 projection='3d'参数，得到一个三维坐标系，如图 3-5-1 所示。

```
In [2]: fig = plt.figure()
        ax = plt.axes(projection='3d')
```

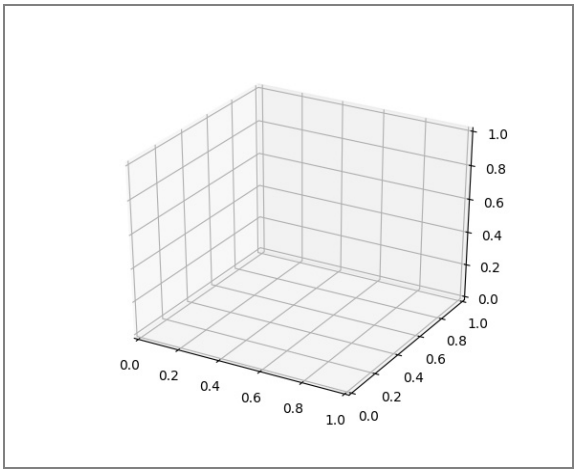


图 3-5-1　三维坐标系

在三维坐标系中，一个点应该通过(*x*, *y*, *z*)来确定，对照二维坐标系中的做法，在三维坐标系中要画出曲线图或者散点图，可以使用 ax.plot3D 和 ax.scatter3D 函数，效果如图 3-5-2 所示。

```
In [3]: ax = plt.axes(projection='3d')

        x_line = np.linspace(0, 15, 1000)
        y_line = np.sin(x_line)
        z_line = np.cos(x_line)
        ax.plot3D(x_line, y_line, z_line, 'blue')

        x_point = 15 * np.random.random(100)
        y_point = np.sin(x_point) + 0.1 * np.random.randn(100)
        z_point = np.cos(x_point) + 0.1 * np.random.randn(100)
        ax.scatter3D(x_point, y_point, z_point, c=x_point, cmap="Greens")
```

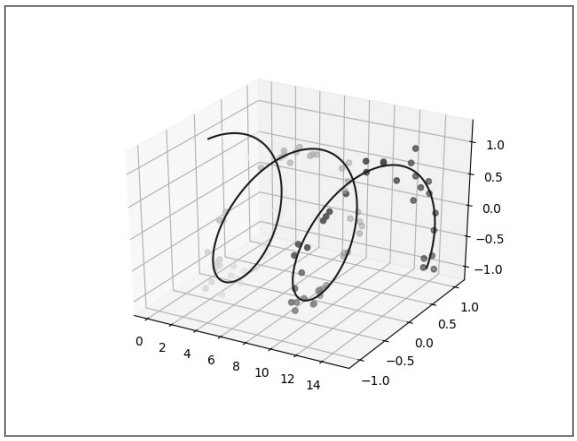


图 3-5-2　三维坐标系中的曲线图和散点

下面以“等高线”为例说明三维图像的具体应用。“等高线”是中学地理课中出现的，不知道读者是否还有印象。如果看它的二维图像，就是“一圈一圈”的线，如果没有特别的训练，跟高度怎么也联系不起来，若改为三维图像（如图 3-5-3 所示），就好多了。

```
In [4]: x = np.linspace(-6, 6, 30)
        y = np.linspace(-6, 6, 30)
        X, Y = np.meshgrid(x, y)    #①
        Z = np.sin(np.sqrt(X**2 + Y**2))

        fig = plt.figure()
        ax = plt.axes(projection='3d')
        ax.contour3D(X, Y, Z, 50, cmap='binary')    #②
        ax.set_xlabel("x")
        ax.set_ylabel("y")
        ax.set_zlabel("z")
        ax.view_init(60, 35)    #③
```

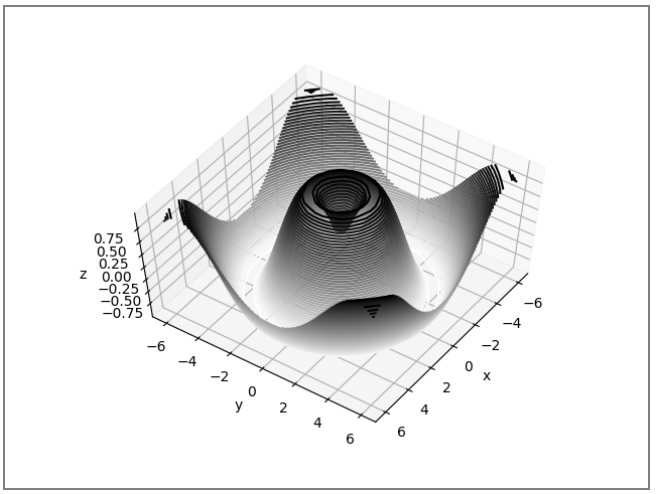


图 3-5-3　三维等高线图

In[4]实现了三维等高线作图，里面有三条语句使用了新函数——对于新函数，我们的研究方法就是看文档。

x 和 *y*（注意字母是小写的）分别是两个一维数组，以其为基础，通过①的 np.meshgrid()函数构建矢量化的对象 ***X*** 和 ***Y***（字母是大写的）。***X*** 和 ***Y*** 就是两组 30×30 并且数值对应的矩阵。因此通过 ***X***、***Y*** 就能构建起平面的网格了。

前面是数据准备，到②这里就是画等高线的函数了。这个函数的参数的具体含义，请读者自行查看文档。

③有必要介绍一下，如果没有这一句，不影响程序执行，只是画出来的图的坐标系像图 3-5-2 那样。要实现图 3-5-3 这样的坐标系，必须使用语句③，它的作用就是实现坐标系方位的旋转。

除等高线外，三维图像还有一个经典应用：绘制莫比乌斯带。

莫比乌斯带（德语：Möbiusband），又可译为梅比斯环或麦比乌斯带，是一种拓扑学结构，它只有一个面（表面）和一个边界，如图 3-5-4 所示。它是由德国数学家、天文学家莫比乌斯（August Ferdinand Möbius）和约翰 • 李斯丁（Johhan Benedict Listing）在 1858 年独立发现的。更详细的内容请阅读《维基百科》相关词条。

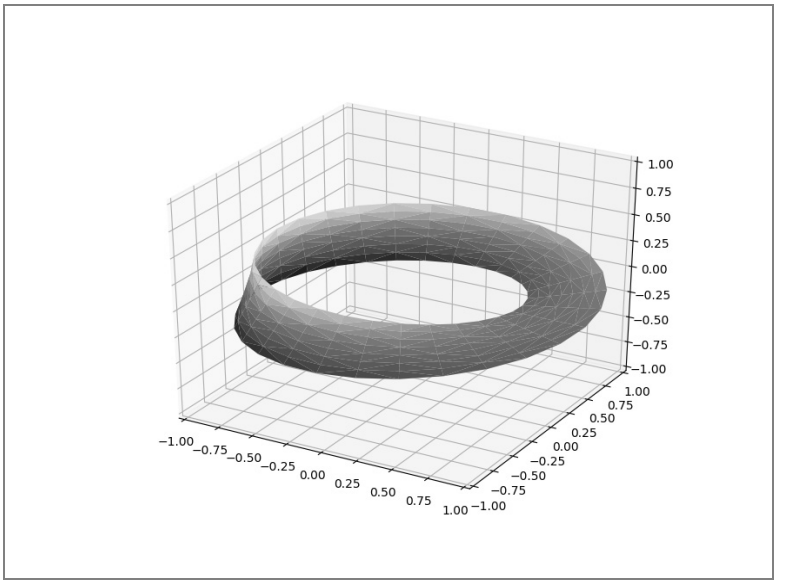


图 3-5-4　莫比乌斯带

绘制莫比乌斯带的基本依据是如下方程组：

$$x(u,v)=(1+\frac{v}{2}\cos\frac{u}{2})\cos(u)$$

$$y(u,v)=(1+\frac{v}{2}\cos\frac{u}{2})\sin(u)$$

$$z(u,v)=\frac{v}{2}\sin\frac{u}{2}$$

（$0\leqslant u\leqslant 2\pi$，$-1\leqslant v\leqslant 1$）

依据此方程组，编写如下程序（参考 *Python Data Science Handbook*）。

```
In [5]: u = np.linspace(0, 2 * np.pi, 30)
        v = np.linspace(-0.5, 0.5, 8) / 2.0
        v, u = np.meshgrid(v, u)

        phi = 0.5 * u

        r = 1 + v * np.cos(phi)
        x = np.ravel(r * np.cos(u))
        y = np.ravel(r * np.sin(u))
        z = np.ravel(v * np.sin(phi))

        from matplotlib.tri import Triangulation     #①
        tri = Triangulation(np.ravel(v), np.ravel(u))    #②

        ax = plt.axes(projection="3d")
        ax.plot_trisurf(x, y, z, triangles=tri.triangles, cmap='viridis',
                        linewidths=0.2)    #③

        ax.set_xlim(-1, 1)
```

```
    ax.set_ylim(-1, 1)
    ax.set_zlim(-1, 1)
```

这段程序基本上是按照前面的方程组写的，重点看几个标识了序号的语句。

①引入了 matplotlib.tri 模块中的 Triangulation 类。matplotlib.tri 模块是干什么的？它是 Matplotlib 库中专门针对非结构化网格作图的模块。这句解释又引入了新的名词“非结构化”“网格”，这真是概念套概念，我们就生活在概念构建的世界中。限于本书的目的，在此不把这些概念展开，如果读者要深入理解，可以依据“拓扑学”“网格”“结构化网格”“非结构化网格”这几个关键词去查找资料。为了能顺利过渡到对语句②的讲述，引用《维基百科》中对“非结构化网格”的解释，即：

非结构化网格是没有规则的拓扑关系的网格，它通常由 polygon triangulation 组成。网格中的每个元素都可以是二维的多边形或者三维多面体，其中最常见的是二维的三角形和三维的四面体。

为此，在 matplotlib.tri 模块中提供了名为 Triangulation 的类，实现元素为三角形的非结构化网格，其完整形式是：

```
Triangulation(x, y, triangles=None, mask=None)
```

下面要插入一大段解释这个类的内容，请读者高度集中注意力，因为解释完之后还要回到②，思路不要乱，这里的写法多少有点“后现代主义”的味道。

```
In [6]: points_xy = np.array([[0.3, 0.5], [0.6, 0.8], [0.5, 0.1], [0.1, 0.2]])   #④
        triangles = [[0, 2, 1], [2, 0 ,3]]   #⑤
        triang = Triangulation(points_xy[:, 0], points_xy[:, 1], triangles=triangles)
        #⑥
        plt.triplot(triang, marker="o")   #⑦
```

In[6]的代码是为了解释 Triangulation()类而写的。请结合图 3-5-5 阅读下述内容并理解代码，注意图 3-5-5 中的序号，是为了讲解方便，后期用作图工具加上去的，运行 In[6]所得结果中不包含序号。

In[6]的④创建了 4 个点的坐标，并用数组表示，可以用索引表示这 4 个点的顺序，如图 3-5-5 中所标识的那样。

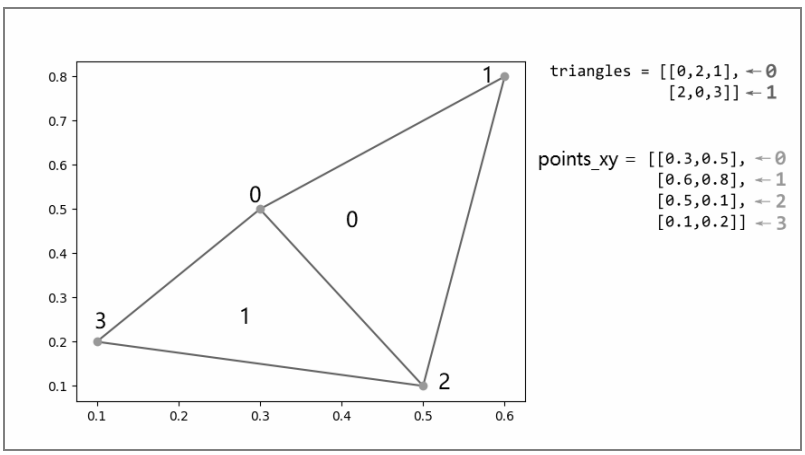


图 3-5-5　非结构网格的三角形

在一个平面中，可以用 4 个点构建两个相邻的三角形——很显然，数学系的朋友非要证明的话，我很欣赏，但此处不做证明。在 In[6]的⑤中就规定了两个三角形分别是用 points_xy 中的哪些点构成的，比如第 0 个三角形，是由第 0 个点、第 2 个点和第 1 个点构成的。从图 3-5-5 还可以看出，构成每个三角形的这些点依据⑤中规定的顺序按逆时针排列，这种排列方式是 Triangulation()类中的 triangles 参数规定的，请看 In[6]的⑥。

在 In[6]的⑦中，triplot()函数是专门用来绘制二维的非结构化三角形网格的函数。

在 In[6]的基础上，如果再增加一个维度，就可以画出三维图像了。

```
In [7]: points_xy = np.array([[0.3, 0.5], [0.6, 0.8], [0.5, 0.1], [0.1, 0.2]])
        triangles = [[0, 2, 1], [2, 0 ,3]]
        triang = Triangulation(points_xy[:, 0], points_xy[:, 1], triangles=triangles)

        z = [0.1, 0.2, 0.3, 0.4]

        ax = plt.axes(projection="3d")
        ax.plot_trisurf(triang, z)    #⑧
```

In[7]与 In[6]的差别在于，定义了另外一个维度 *z*，然后用⑧基于三角形网格绘制三维图像，得到了如图 3-5-6 所示的效果。

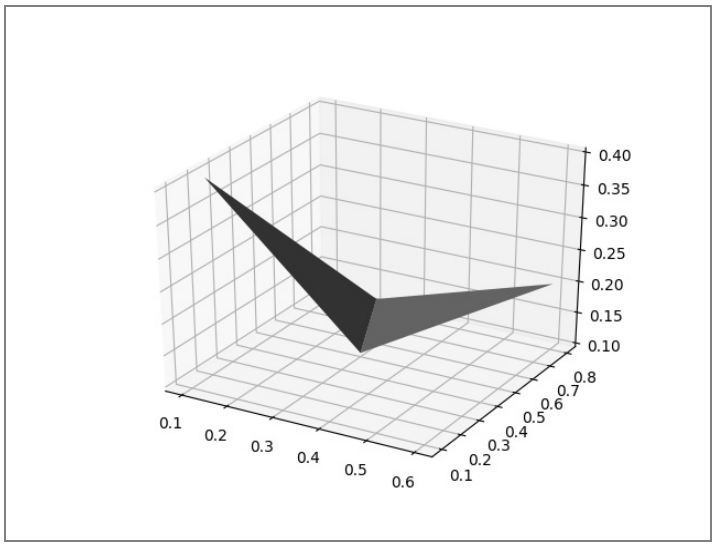


图 3-5-6　三维的三角形网格

在 In[6]和 In[7]的基础上，回头再看 In[5]的①②③，就一目了然了。

所以，遇到有点难度的东西，最好的解决方法就是将它分解，先研究局部，然后组合起来就弄懂了。

Matplotlib 可以绘制三维图像，但术业有专攻，在这个领域，还有一些专门的工具，比如 TVTK、Mayavi 等。如果读者需要专业化地绘制三维图像，建议对这些专门库进行学习。当然，Matplotlib 也是可以使用的。

3.6　Seaborn 掠影

在前面的某章某节，已经安装了 Seaborn，曾记否？

Matplotlib 已经是一个相当不错的、跨平台、功能强大的绘图工具了，并且一直在与时俱进。但是因为历史原因，它还有诸多不尽人意之处——历史悠久的，都有一些或好或坏的传统。

于是，Seaborn 应运而生。

Seaborn 基于 Matplotlib，且“青取之于蓝而胜于蓝”。

下面以示例的方式展示 Seaborn 的用法。篇幅所限，如果读者要系统了解，请阅读官方文档。在学习本章前述各节的基础上，掌握 Seaborn 是小菜一碟。这就好比张无忌已经有了乾坤大挪移等绝世武功基础之后，再学习张三丰的太极拳，就能一下抓住其要领，而不在乎具体招式，短时间就能运用自如——精通某个技能之后，可以顺利迁移。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import seaborn as sns
        sns.set()
In [2]: iris = sns.load_dataset("iris")
        iris.head()
Out[2]:
          sepal_length  sepal_width  petal_length   petal_width  species
        0          5.1          3.5           1.4           0.2   setosa
        1          4.9          3.0           1.4           0.2   setosa
        2          4.7          3.2           1.3           0.2   setosa
        3          4.6          3.1           1.5           0.2   setosa
        4          5.0          3.6           1.4           0.2   setosa
```

在 2.7 节中，我们曾经使用类似 In[2]的方式读取了泰坦尼克号的有关数据，这里则读取了 Seaborn 中集成的鸢尾花数据——此数据是数据分析和机器学习中的经典数据集。并且从输出结果中可以看出，它是 DataFrame 类型。

```
In [3]: sns.pairplot(iris, hue='species', size=2.5)
```

结合图 3-6-1，可知 In[3]的目的在于依据“species”列的分类，分别绘制反映每两个字段（视为变量）之间数据关系的散点图。读者有兴趣的话，可以仔细看看这张图，因为后面还要对相关变量值进行分析。

In[3]所使用的 pairplot()方法完整结构是：

```
seaborn.pairplot(data, hue=None, hue_order=None, palette=None, vars=None, x_vars=None,
y_vars=None,  kind='scatter',  diag_kind='hist',  markers=None,  size=2.5,  aspect=1,
dropna=True, plot_kws=None, diag_kws=None, grid_kws=None)
```

其中 data 就是一个 DataFrame 类型的数据集，每一列被视为一个变量。在 In[3]中还使用了另外一个参数 hue，指定了用于分类的变量（data 的列）。其他参数的含义请自行查看文档。

In[3]是 Seaborn 小试牛刀之作，即能看出其操作简便了。

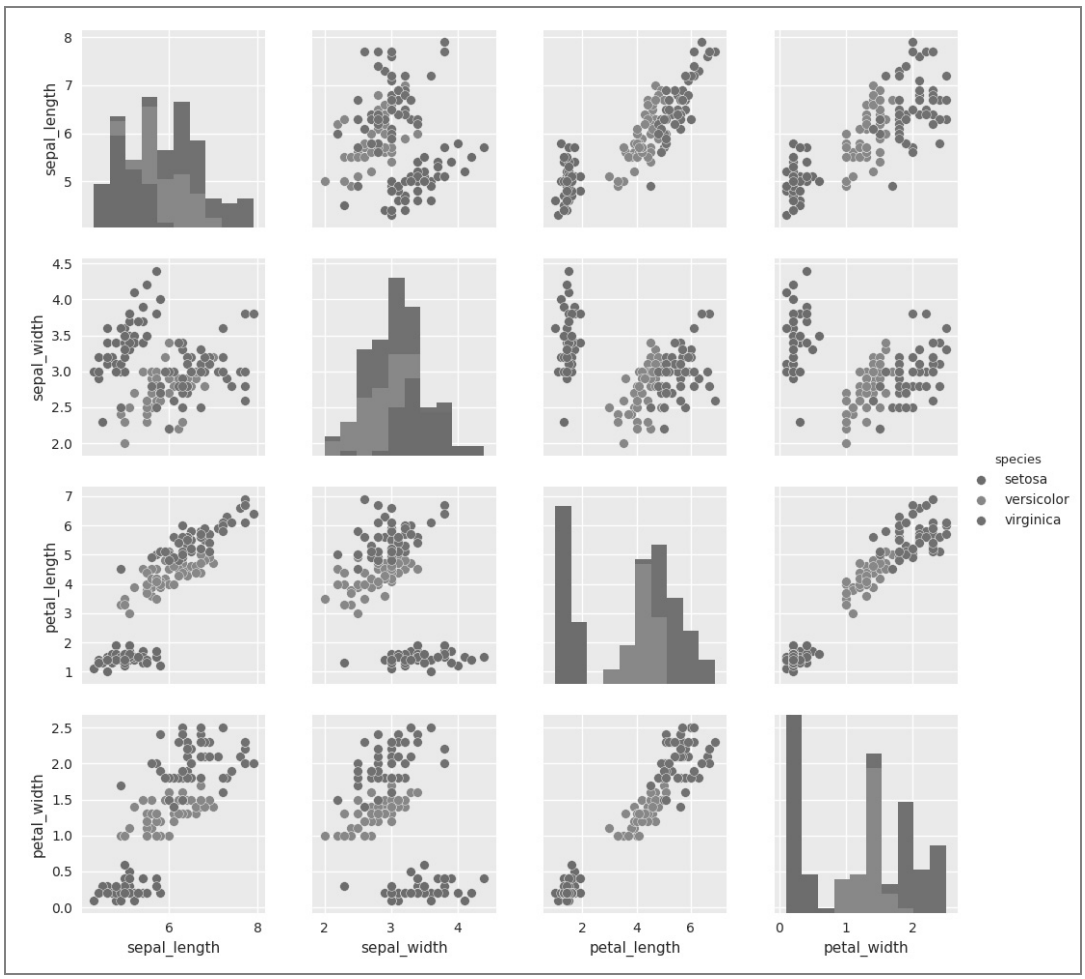


图 3-6-1　指定变量，比较数据集字段

在 3.4 节曾经绘制过箱线图，这里再用 Seaborn 绘制这种图，如图 3-6-2 所示。可以跟前面的绘制过程进行比较。

```
In [4]: import matplotlib.pyplot as plt
        planets = sns.load_dataset("planets")    #①

        f, ax = plt.subplots(figsize=(7, 6))
        ax.set_xscale("log")

        sns.boxplot(x="distance", y="method", data=planets, whis=np.inf,
                    palette="vlag")    #②
        sns.swarmplot(x="distance", y="method", data=planets, size=2, color=".3",
                      linewidth=0) #③
        ax.xaxis.grid(True)
        ax.set(ylabel="")
        sns.despine(trim=True, left=True)
```

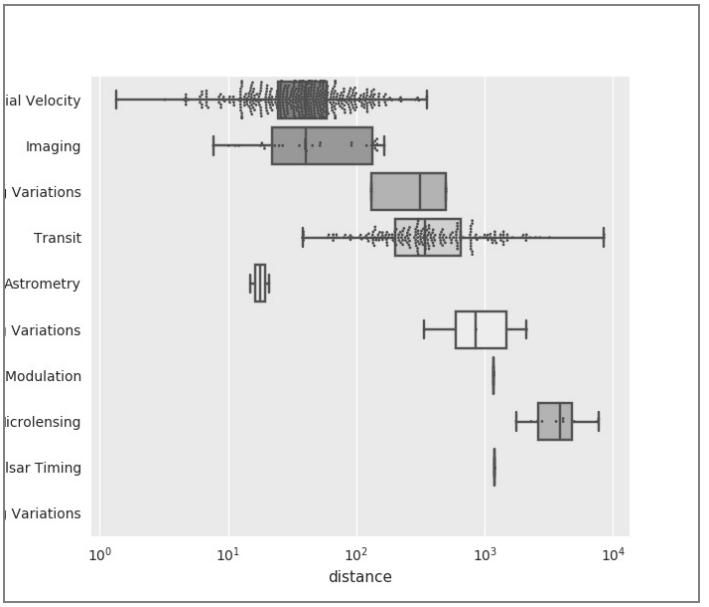


图 3-6-2　箱线图

In[4]的①获取了一组新的数据集，随后在②中使用 boxplot()函数绘制此数据集的箱线图。

```
boxplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None, orient=None, color=None, palette=None, saturation=0.75, width=0.8, dodge=True, fliersize=5, linewidth=None, whis=1.5, notch=False, ax=None, **kwargs)
```

虽然参数列表中显示的参数很多，但依据以往的经验，没有必要去记忆，只需要知道如何查看帮助文档即可，文档中对每个参数都做了解释。这里仅对②中几个常用的参数予以简单说明。

- *x*、*y*：规定横、纵坐标的变量（以数据集的列名称作为变量）。
- data：绘制图像所需的数据集。
- whis：在绘制箱线图的时候，如果数值超出了最高和最低四分位数，在图示中需要向两侧分别延伸（即线的长度），whis 规定了延伸的最大范围。对于超过此范围的数值，都被视为异常值。在②中，whis 的值 np.inf 表示一个无限大的数。

In[4]的③为另外一个绘图函数，它的作用是绘制不重叠的分类散点图，通过这类图，能够观察到数值的分布特点。因为数值本身的特点，往往会集中在某个区域，所以分布得很“拥挤”，像蜜蜂拥挤在蜂巢上一样，所以此函数名为 swarmplot()，下面是它的完整参数列表。

```
swarmplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None, dodge=False, orient=None, color=None, palette=None, size=5, edgecolor='gray', linewidth=0, ax=None, **kwargs)
```

除绘制箱线图外，Seaborn 也可以用来绘制 3.4 节中所介绍的其他常用统计图，此处不一一介绍了。下面再看一个前面没有遇到过的统计图，如图 3-6-3 所示。

```
In [5]: rs = np.random.RandomState(0)
        n, p = 40, 8
        d = rs.normal(0, 2, (n, p))
        d += np.log(np.arange(1, p + 1)) * -5 + 10
```

```
pal = sns.cubehelix_palette(p, rot=-.5, dark=.3)    #①

sns.violinplot(data=d, palette=pal, inner="points")    #②
```

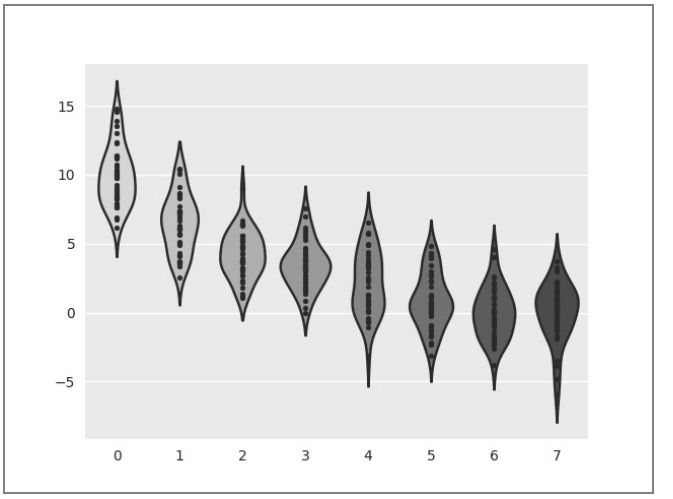


图 3-6-3　小提琴图

看样子很奇怪。

这个怪模样的图名字倒是很文雅——小提琴图（Violin Plot），这种图融合了箱线图和密度图的特征，主要用来显示数据的分布形状。

In[5]的①是一个颜色控制函数，图 3-6-3 中的各个小提琴图的颜色就是按照此处所规定的序列显示的，这个显示顺序来自于 cubehelix 系统——这是绘图中的一款调色板，它所产生的色彩亮度能够按照一个线性增加（或减少）的顺序变化。

In[5]的②用于绘制小提琴图，其参数的完整列表建议读者查看文档，此处不再列举。

对于 Seaborn 而言，可以绘制的图像有很多，在官方网站（https://seaborn.pydata.org/examples/）有相当多的示例，读者可以在空闲的时候查看并调试。

本节的标题就已经声明，这里仅仅是 Seaborn“掠影”，亮个相。如果读者有需要，凭借在本章所学的本领，能够很快运用自如——就是这么自信，理当如此。并且，本章开篇就说明了，可以实现数据可视化的工具还有很多。不过读者不用慌，前面已经列举了一个榜样——张无忌。学好了本章的工具，特别是 Matplotlib，再学别的工具也会很快——阅读此文字的读者就是正在崛起的数据分析和机器学习大侠。

大侠的初长成，要历经磨难。最终把各类武功融会贯通，做到信手捏来，草木皆剑——还有比这厉害的呢。不论怎样，都要艰苦奋斗。

第4章

综合应用

前面几章“磨刀霍霍”，本章则“向猪羊”，是否能够游刃有余？

李寻欢的刀，例不虚发；

傅红雪的刀，活人看不到；

《庄子》中的解牛刀，游刃有余。

在夕阳下。

他本就是完全孤独的。

“他下过苦功，据说他每天至少要花四个时辰练刀，从四五岁的时候开始，每天就至少要拔刀一万两千次。”

那把刀，已经不仅是一把刀了，他的人与刀之间，已经有了一种别人无法了解的感情（古龙《天涯·明月·快刀》）。

江湖公认为天下第一快刀——傅红雪，领悟到了“梅花香自苦寒来”。

与之相比，我们何如？

笑傲江湖，可否？

“忘记背后，努力面前”，你我共勉。

4.1 分析股票数据

很多人梦想着通过买卖股票发财，虽然未必成真，却在不断追求。既然如此，“炒股界”必然会集中一些聪明的头脑，他们会用各种自认为高超的方法来分析如何买卖股票，比如各路大师不断兜售着各种方法——他们是否都用那些方法发财了？

真正的股市，变幻莫测。

所以，才吸引人来探索其中的规律。

下面就来看看我们的“刀法”在股市中如何使用。

1．股价波动曲线

曾记否，在 2.10 节中，我们已经读取了股票价格，并且尝试着作了图。

注意，跟前述原因一样，因不可抗力，无法保证按照如下方式就能读取到相应数据。但是，如果读者愿意在网上搜索，或许可以找到解决方案。

```
In [1]: import pandas as pd
        import matplotlib.pyplot as plt
        import pandas_datareader

        aapl = pandas_datareader.DataReader("AAPL", 'google')
        goog = pandas_datareader.DataReader("GOOG", 'google')
        baba = pandas_datareader.DataReader("BABA", 'google')
        amzn = pandas_datareader.DataReader("AMZN", 'google')
```

先把数据读到内存中，如果遇到问题，请参考 2.10 节中的说明。

然后把各只股票的收盘价组合成为一个 DataFrame 对象，并画图。

```
In [2]: aapl_close = aapl['Close']
        goog_close = goog['Close']
        baba_close = baba['Close']
        amzn_close = amzn['Close']

        stocks = pd.DataFrame({'APPL':aapl_close,
                               'GOOG':goog_close,
                               'BABA':baba_close,
                               'AMZN':amzn_close})
        stocks.plot()
```

我们喜欢看图，所以会经常将数据可视化。

图 4-1-1 反映的是在 In[1]中选择的 4 只股票收盘价随时间的变化情况。这个图的优点在于绘制简单，并且直观地显示了股票价格的变化。

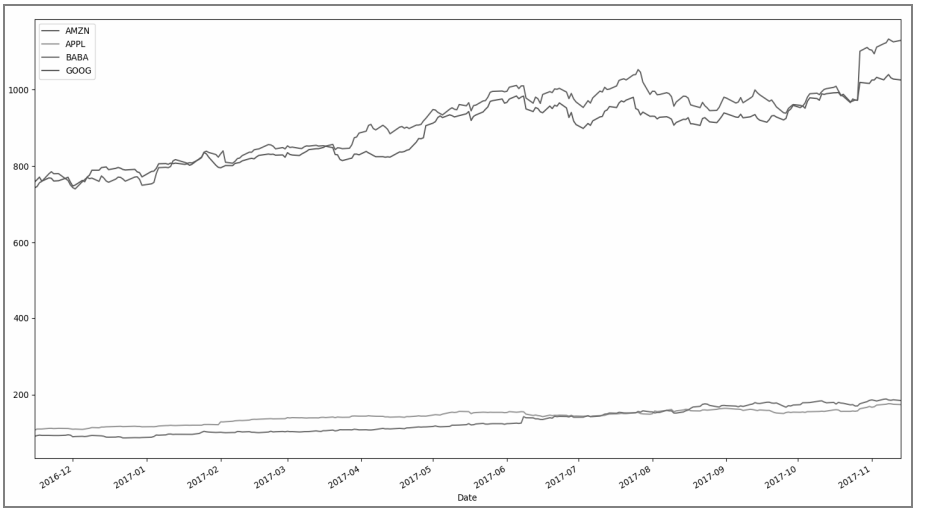


图 4-1-1　部分股票收盘价的时间序列图

不过，从图中也可以明显地看出，不同股票的价格不一样。如果要买入，买哪个？买价格低的吗？非也。

股民们告诉我，他们很少看这个图，他们更多地是看“K 线图”。

2．K 线图

K 线图（Candlestick Chart），是反映价格走势的一种图线，其特点在于一个线段内记录了多项信息，易读易懂且实用有效。图 4-1-2 是依据《维基百科》的“K 线”词条绘制的，供读者参考。

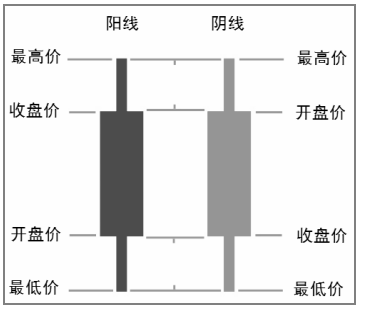


图 4-1-2　K 线的含义

据说 K 线图的颜色代表着涨跌含义，而且中国的和欧美的还反着——真够乱的，这就不如代码，世界大同。

不炒股，水太深，还是敲代码，学习 K 线图怎么画出来吧。

```
In [3]: from matplotlib.dates import DateFormatter, WeekdayLocator, DayLocator, MONDAY
        from matplotlib.finance import candlestick_ohlc    #①
        import matplotlib.dates as mdates    #②
        import datetime

        start = datetime.datetime(2017, 10, 1)
        end = datetime.datetime(2017, 11, 13)
        baba_se = baba.loc[start: end]    #③

        mondays = WeekdayLocator(MONDAY)    #④
        alldays = DayLocator()
        week_formatter = DateFormatter("%b %d")

        fig, ax = plt.subplots()
        fig.subplots_adjust(bottom=0.2)
        ax.xaxis.set_major_locator(mondays)    #⑤
        ax.xaxis.set_minor_locator(alldays)
        ax.xaxis.set_major_formatter(week_formatter)

        baba_ohlc = baba_se.reset_index()
        baba_ohlc['Date'] = baba_ohlc['Date'].map(mdates.date2num)    #⑥
        candlestick_ohlc(ax, baba_ohlc.values, width=0.7, colorup='r', colordown='g',
                         alpha=0.7)  #⑦
```

```
    ax.plot(baba_se['Close'])    #⑧

    ax.xaxis_date()
    ax.autoscale_view()
    plt.setp(plt.gca().get_xticklabels(), rotation=45, horizontalalignment='right')
    plt.grid(True)
```

In[3]的程序绘制出了图 4-1-3 所示的 K 线图——股民熟悉的图。笔者不熟悉，只能解释代码，不能解释图中所展现的深层次含义，代码中的序号含义如下。

图 4-1-3　K 线图

① 引入的是 matplotlib.finance 模块中的一个方法 candlestick_ohlc()，K 线图又称为蜡烛图、棒线图，从这个函数名称中就可以看到这两个名称的影子。

③ 表示选择了“BABA”中的某个时间段内的数据。

④ 及其以下的两句，分别确定了坐标轴上的刻线和标示对象。

⑤ 及其以下的两句，具体设置了 *X* 轴的主、副刻线和标示。

重点在⑦，使用 candlestick_ohlc()函数绘制 K 线图，此函数的主要参数如下。

- ax：毋庸置疑，就是所创建的 Axes 对象，前面的“fig, ax = plt.subplots()”所得到 ax 引用的对象即是。
- quotes：包含 time、open、high、low、close 序列，在⑦中传入的是一个 DateFrame 对象。函数名称中的 ohlc，依次按照 open、high、low、close 的顺序一一对应。还要注意其 time 必须是以日（天）的浮点数格式表示的，为此使用②，并且在⑥中使用 mdates.date2num 对 datetime 类型的日期进行转换。
- width：矩形的宽度，默认 0.2。
- colorup：默认为“k”（黑色），用于设置股价上升的颜色。colorup='r'遵守了我国的习惯。
- colordown：默认为“r”（红色），用于设置股价下跌的颜色。colordown='g'也是按照我国的习惯设置的。

为了能够对比理解，再以⑧画出相应的收盘价曲线。

有了 K 线图，再回头看图 4-1-1，能确定买哪只股票了吗？

还不能。因为炒股的财主们除关注股票的价格外，还要关注股票的变化，即相对于某个时间段股票的涨跌及其数量，这就是“相对变化”。

3．相对变化

一种比较简单的计算相对变化的方法就是先确定一个初始量，然后用以后某个时刻的量除以初始量。以前面“BABA”的股票为例，如果将 2016 年 11 月 15 日的股票价格作为初始值，可以计算以后每一天的股票价格相对此值的比例，这个比例显然是一个相对值，如图 4-1-4 所示。

```
In [4]: baba_return = baba.apply(lambda x: x / x[0])
In [5]: baba_return.head()
Out[5]: Date              Open        High         Low       Close       Volume
        2016-11-15    1.000000     1.000000    1.000000    1.000000    1.000000
        2016-11-16    1.004286     1.002032    1.006391    1.020440    0.841936
        2016-11-17    1.040769     1.015938    1.031846    1.033736    0.620273
        2016-11-18    1.042308     1.019360    1.021157    1.026264    0.599928
        2016-11-21    1.033516     1.012515    1.024793    1.026484    0.535793

In [6]: baba_return[-5:]
Out[6]: Date              Open        High         Low       Close       Volume
        2017-11-07    2.083297     2.030805    2.060606    2.071538    1.063267
        2017-11-08    2.065714     2.012515    2.039669    2.042857    0.833859
        2017-11-09    2.024286     1.980212    1.995041    2.034396    1.181343
        2017-11-10    2.055275     2.002032    2.042975    2.048462    0.898824
        2017-11-13    2.052418     2.009627    2.027548    2.027912    1.027666

In [7]: baba_return[['Open', 'Close']].plot(grid=True).axhline(y = 1, color = "black",
                                                               lw = 2)
```

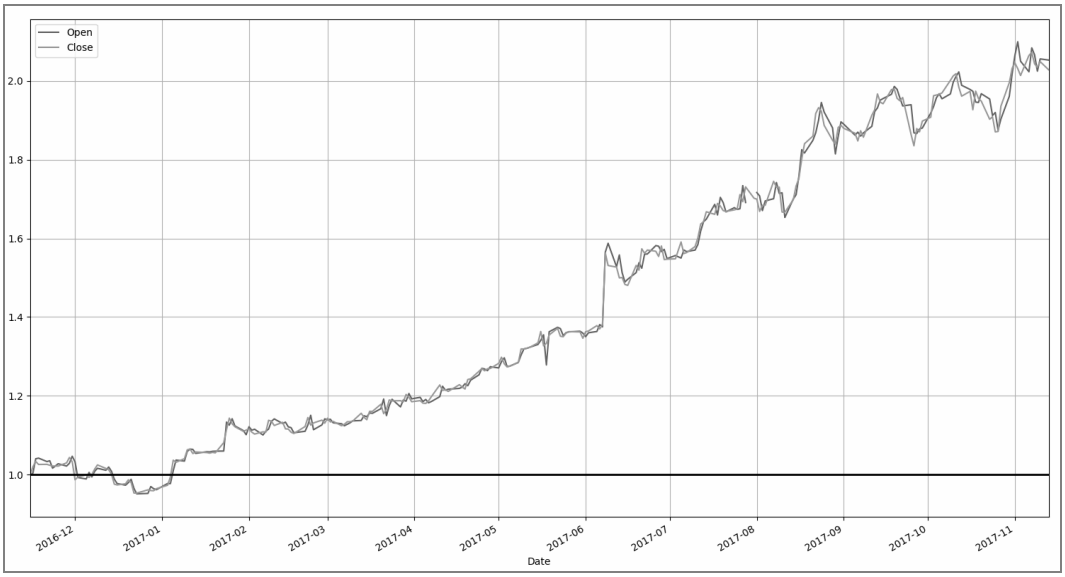


图 4-1-4　开盘价和收盘价相对初始值的比例

从计算结果和图示结果都可以看出，如果在 2016 年 11 月 15 日购买了“BABA”股票，到上述统计结束日期（2017 年 11 月 13 日）就赚翻了——恭喜发财。

此外，还有另外一种计算相对变化的方法，就是计算每天相对于前一天的变化率。基本公式是：

$$\text{change} = \frac{price_{today} - price_{last}}{price_{last}}$$

注意，如果读者是股票方面的行家，则不要在上面的公式上较真了。据笔者有限的知识，在股市中还区分“涨幅”和“跌幅”，计算方法也有两种。这里统一按照物理学中对物理量变化率的规定来定义“股票的变化率”。

依据《维基百科》中的“Rate of return”词条解释，在金融学中，“收益率”的定义为：

$$r = \frac{V_f - V_i}{V_i}$$

现在理解为什么那么多学物理的人后来去华尔街了。

写个函数，实现上面的公式吧。

且慢。物理是“普适”的，是不是应该有专门的函数实现上面的变化率运算呢？

必须有，效果如图 4-1-5 所示。

```
In [8]: import datetime
        baba_close_change = baba['Close'].loc[datetime.datetime(2017,10,1):].pct_change()
        amzn_close_change = amzn['Close'].loc[datetime.datetime(2017,10,1):].pct_change()
        df = pd.DataFrame({"BABA":baba_close_change, "AMZN":amzn_close_change})
        df.plot(grid=True) .axhline(y=0, color='black', lw=2)
```

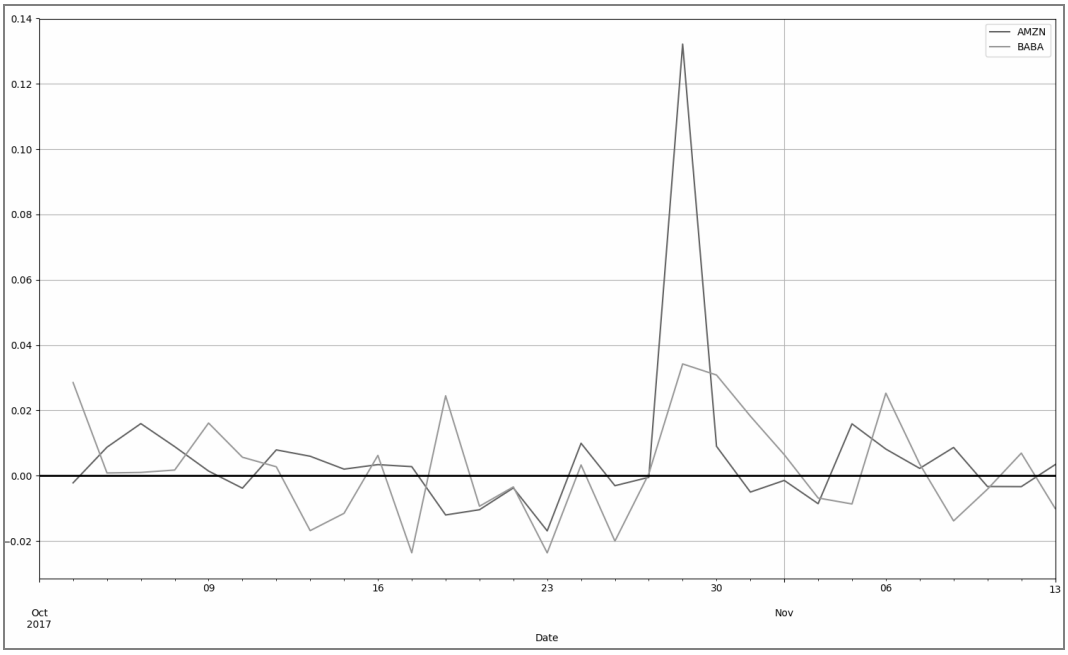


图 4-1-5　BABA 和 AMZN 两只股票每日收益率折线图

```
In [9]: baba['Close'].loc[datetime.datetime(2017,10,1):].head()
Out[9]: Date
        2017-10-02    173.61
        2017-10-03    178.56
        2017-10-04    178.71
        2017-10-05    178.89
        2017-10-06    179.20
        Name: Close, dtype: float64

In [10]: baba_close_change.head()
Out[10]: Date
         2017-10-02        NaN
         2017-10-03    0.028512
         2017-10-04    0.000840
         2017-10-05    0.001007
         2017-10-06    0.001733
         Name: Close, dtype: float64
```

除如上计算“收益率”的方式外，还有另外一种通过对数计算的方式，目的都是找出相对变化，如图 4-1-6 所示。

```
In [11]: import numpy as np
         amzn_close_part = amzn['Close'].loc[datetime.datetime(2017,10,1):]
         amzn_close_change = np.log(amzn_close_part) - np.log(amzn_close_part.shift(1))
         amzn_close_change.plot(grid=True).axhline(y=0, color='k', linewidth=2)
```

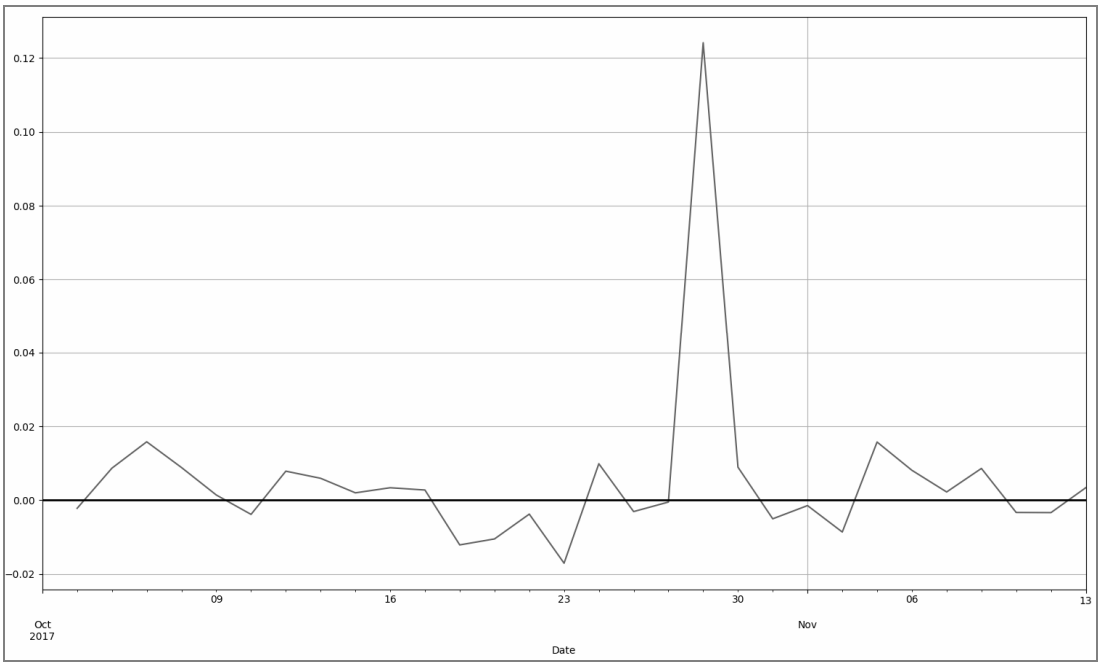


图 4-1-6　利用对数计算的相对变化

以上几种计算相对变化的方法，哪一个对买卖股票有用笔者无法评价，还是由股民定吧。

下面继续从统计学角度研究股票价格。

4．相关分析

分析过去的股票数据，希望能够找到规律，从而预测它的未来——历史选择未来。

这当然是理想了，未来充满了不确定性。

此外还有另外一种方法，就是“看彼知此”，最经典的就是“××的今天就是××的明天”（《江苏教育》1955 年 22 期，网址是 http://www.cnki.com.cn/Article/CJFDTOTAL-JAOI195522008. htm）。

如果用统计学的方法来研究，这是在分析两个变量——彼、此——之间的相关关系，以相关关系为基础，从而确定某个量的未来变化，如常常被津津乐道的经济学中的“裙摆指数”。

下面将这个思想用在股票数据分析中，看看是否可以让你发财。

```
In [12]: pd.plotting.scatter_matrix(amzn)
```

从图 4-1-7 中可以很直观地看出某些量之间的相互关系。比如开盘价和收盘价之间就是正相关关系，从统计学的角度看，开盘价较前一天涨了，当天的收盘价也会涨。

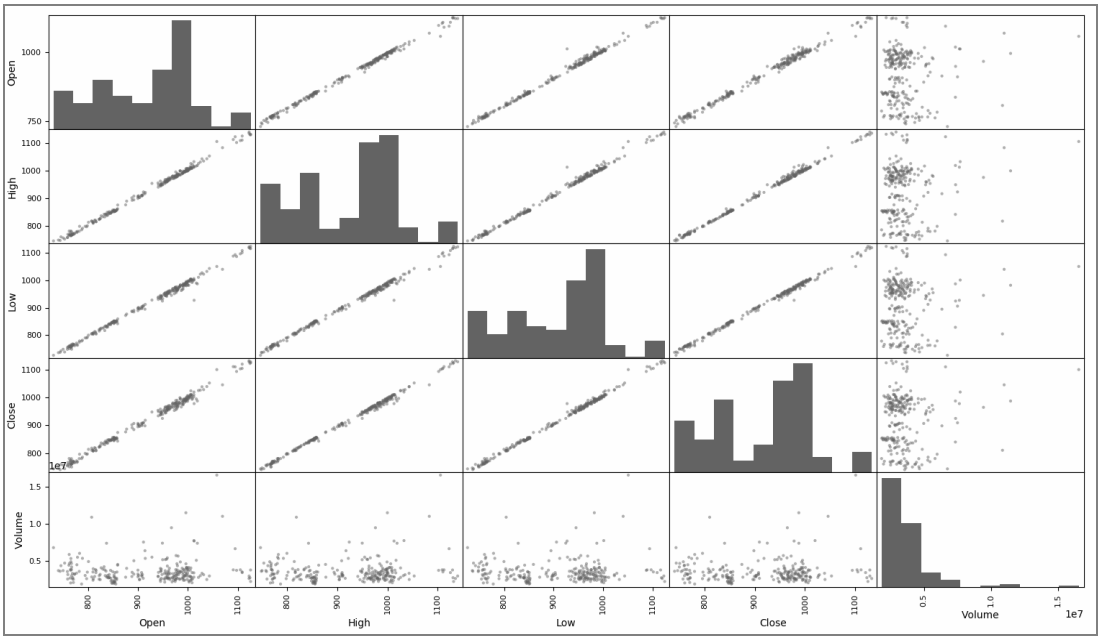


图 4-1-7　相关性散点图

In[12]仅仅是查看相关性的一种方法，或许是巧合，从直观的角度看，各量之间的相关性非常明显。但并非总是如此，为此就要从定量的角度来衡量相关性。

在统计学中，可以用协方差（COV）和相关系数（corrcoef）来衡量相关程度（具体计算方法请参考有关统计学资料）。

```
In [13]: np.corrcoef(amzn.T)
Out[13]: array([[ 1.        ,  0.9981258 ,  0.99684724,  0.99564519,  0.05527373],
                [ 0.9981258 ,  1.        ,  0.9967958 ,  0.99818172,  0.08239939],
                [ 0.99684724,  0.9967958 ,  1.        ,  0.99791858,  0.02851312],
                [ 0.99564519,  0.99818172,  0.99791858,  1.        ,  0.05867276],
                [ 0.05527373,  0.08239939,  0.02851312,  0.05867276,  1.        ]])
```

提供计算相关系数的函数是理所当然的，还有计算协方差的函数 np.cov()。

将计算结果和图 4-1-7 对照，“数”“形”一致。

还可以更直观地反映 Out[13]的数据，如图 4-1-8 所示。

```
In [14]: corr = np.corrcoef(amzn.T)
         img = plt.matshow(corr)
         plt.colorbar(img, ticks=[0, 1])
```

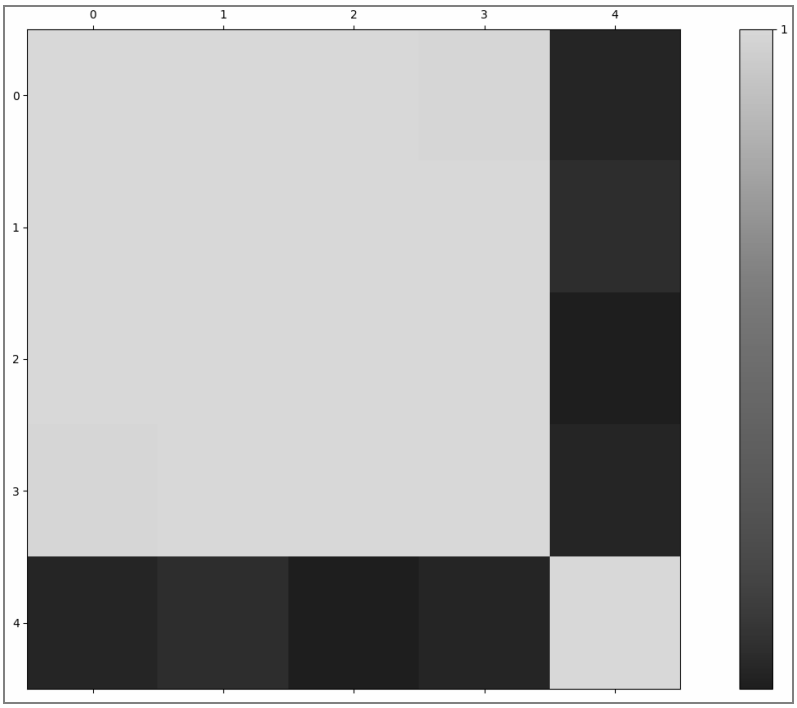


图 4-1-8　相关系数可视化

在本书中，读者看到的是黑白图像，还是反复建议读者要在计算机上调试，才能体会到数据可视化的魅力。

5．决策时机

何时买？何时卖？

以上的各项分析固然有一些参考价值，但股市风云变幻，所以人们也在不断发明新的指标协助决策。

还记得“移动平均线”吗？这是一个很重要的指标，此处不能忽略。

```
In [15]: baba_train = baba[datetime.datetime(2017,6,1):datetime.datetime(2017,8,1)]
         baba_train_close = baba_train['Close']
         ma20_mean = np.round(baba_train_close.rolling(window=20, center=True).mean(), 2)
         ma5_mean = np.round(baba_train_close.rolling(window=5, center=True).mean(), 2)
         fig, ax = plt.subplots()
         ax.plot(ma5_mean, label="MA5", linestyle="-")
         ax.plot(ma20_mean, label="MA20", linestyle="--")
         plt.legend()
```

```
        plt.grid(True)
```

In[15]计算了 MA20 和 MA5 并分别绘线，从图 4-1-9 可以明显看出，MA20 的曲线相对更平滑。据“炒股界分析和预测并指导买入卖出的股神大师”（以下简称“大师”）说，移动平均线具有抹平短期波动的作用，更能反映长期的走势。并且，“大师”特别关注 MA20 和 MA5 的交叉点，他认为那里是交易的时机，当 MA5 均线从下方超越 MA20 均线时，买入股票；当 MA5 均线从上方跌到 MA20 均线之下时，卖出股票。

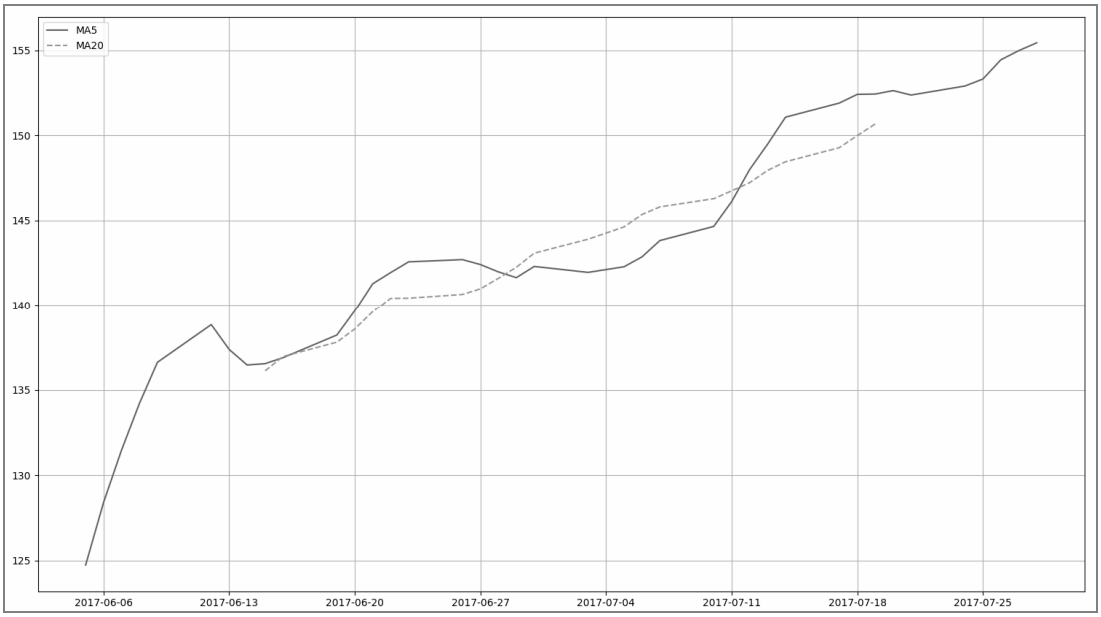


图 4-1-9　BABA 的 MA5 和 MA20 的图线

结合图 4-1-9 来看，貌似“大师”的分析有理。

那就按照“大师”的理论继续进行计算，目标是选出适合交易的良辰吉日。

```
In [16]: baba_diff = ma5_mean - ma20_mean    #①
         signal = np.sign(baba_diff - baba_diff.shift(1))    #②
         buy = pd.DataFrame({"price": baba_train.loc[signal.values==1, "Close"],
                             'operation': "Buy"}) #③
         sell = pd.DataFrame({"price": baba_train.loc[signal.values==-1, "Close"],
                              'operation': "Sell" })#④
         trade = pd.concat([buy, sell])
         trade.sort_index(inplace=True)
         trade
Out[16]: Date       operation   price
         2017-06-16   Sell     134.87
         2017-06-19   Buy      139.47
         2017-06-20   Buy      138.38
         2017-06-21   Buy      143.29
         2017-06-22   Sell     142.27
         2017-06-23   Buy      143.01
         2017-06-26   Sell     142.73
         2017-06-27   Sell     141.53
```

```
2017-06-28   Sell     143.95
2017-06-29   Sell     140.81
2017-06-30   Sell     140.90
2017-07-03   Sell     140.99
2017-07-05   Sell     144.87
2017-07-06   Sell     142.20
2017-07-07   Buy      142.43
2017-07-10   Buy      143.81
2017-07-11   Buy      145.81
2017-07-12   Buy      149.00
2017-07-13   Buy      149.52
2017-07-14   Buy      151.83
2017-07-18   Sell     153.75
2017-07-19   Sell     153.15
```

In[16]的代码还是依照“大师”的指导写的。①计算 MA5 和 MA20 的差，然后计算 baba_diff 相邻日期的差值，并用 np.sign()函数得到信号指标，即为②的含义。当 signal=1 时，表示此时要买入股票；当 signal=-1 时，表示此时应该卖出股票；当 signal=0 时，就要静观其变。

这样，“大师”就构造了一个买卖股票的模型。此模型是否管用？

③④两句就是把所有买入和卖出的日期及其收盘价找出来。看一下输出结果，会发现有赔有赚。

这不是“大师”的错，本来就是“股市有风险，入市需谨慎”，更何况，上面的模型还是非常简单的。

复杂的模型就能预测准确了吗？

未必。

不过本书不讨论这个问题。这里仅仅是以对股票数据的分析为例，练习如何综合应用已学知识。

据传，股票界江湖也派系林立，有“技术派”“内幕派”“跟风派”等。

你站哪一派？

4.2　分析文胸评论数据

数据分析，首先要有数据，即解决数据来源问题。4.1 节分析的股票数据来源于 Pandas，还有数据来源于其他处，包括某人/机构整理的 API 等。此外，众多的各种类型的网站是更广阔的数据源。

每个网站都有若干网页，网页上的内容就是我们要获取的数据。从网站上获得数据的方法有多种，比如有的网站提供访问本站数据的 API，但这类网站较少，并且 API 还有诸多限制，所以获取数据的一种技术——“网络爬虫”就得到了广泛应用。

本节就从某东网站上爬取了一点点数据，仅供研习之用。

当然，因为本书不是专门讲解爬虫程序的，所以关于爬虫程序的内容略去了。更何况仅仅是为了演示如何分析数据，所写的爬虫程序也相当简陋，就不在此卖弄了。如果读者对爬虫有

兴趣，推荐到 itdiffer.com 网站查看完整的爬虫学习内容。

读者可以到本书代码仓库中获得本节所用数据。

下面直接进入对数据的分析过程。

首先要对获得的原始数据进行一番整理——俗称“清洗数据”。

```
In [1]: % matplotlib
        import pandas as pd
        import matplotlib.pyplot as plt
        bras = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter04/bra.csv")
        bras.head()
Out[1]:            creationTime    productColor    productSize
        0    2016-06-08 17:17:00        22 咖啡色            75C
        1    2017-04-07 19:34:25        22 咖啡色            80B
        2    2016-06-18 19:44:56          02 粉色            80C
        3    2017-08-03 20:39:18        22 咖啡色            80B
        4    2016-07-06 14:02:08        22 咖啡色            75B
```

在“清洗数据”之前，先看看是什么数据。从上面 Out[1]的输出结果中，读者一定能明白数据所描述的对象是什么了——来自某东网站的部分文胸购买数据。

再看看 productColor 的种类。

```
In [2]: bras['productColor'].unique()
Out[2]: array(['22 咖啡色', '02 粉色', '071 蓝色', '071 黑色', '071 肤色', '0993 无痕肤色',
                '0993 无痕黑色', '071 红色', '0993 无痕酒红色', 'h03 无痕蓝灰', '蓝灰色',
                '酒红色', '内裤酒红色', '内裤蓝灰色', nan, '肤色', '藕荷色', '藕荷色(套装)',
                '玫瑰色', '深蓝色', '烟灰紫', '天蓝色', '黑色', '红色', '香妃红',
                '杏肤色', '诱惑黑', '土豪金', '大红', '嫣紫', '皇家蓝', '粉晶色',
                '柔肤色', '蓝灰', '大红色', '紫色无钢圈厚杯', '肤色无钢圈厚杯',
                '黑色无钢圈厚杯', '紫色套装', '爱心大红色', '肤色套装', '爱心肤色',
                '爱心黑色', '紫色 单件', '紫色 套装', '肤色 单件', '黑色 单价',
                '肤色 套装', '黑色 套装', '浅紫色', '内裤黑色', '内裤肤色', '浅蓝', '浅黄',
                '宝蓝色', '酒红', '灰色', '浅粉色', '醇黑', '红条纹', '蓝色', '蓝条纹',
                'PNK', 'LAV', 'BLK', 'GRN', '浅紫色', '肤色', '浅紫色', '肤色', '枚红色',
                '粉色', '蓝色', '西瓜红', '银灰色', '天蓝色', '8626 黑色', '红色',
                '国色天香 红色【厚】', '磁石款  黑色【薄】', '磁石款  肤色【薄】',
                '磁石款 肤色【薄】', '美背款 粉色【厚】', '磁石款  粉色【厚】',
                '经典款 粉色【薄】', '磁石款 黑色【薄】', '磁石款 肤色【厚】',
                '国色天香 黑色【厚】', '磁石款  红色【厚】', '磁石款 粉色【薄】',
                '天衣无缝 黑色【AB 厚,C 薄】', '美背款 黑色【厚】', '磁石款  黑色【厚】',
                '国色天香 深紫【厚】', '经典款 粉色【厚】', '经典款 肤色【厚】',
                '磁石款  蓝色【厚】', '经典款 黑色【厚】', '经典款 黑色【薄】',
                '国色天香 粉色【厚】', '天衣无缝 粉色【AB 厚,C 薄】', '经典款 肤色【薄】',
                '天衣无缝 肤色【AB 厚,C 薄】', '粉红色', '碧绿色', '嫩黄色', '紫兰',
                '粉色 单件', '粉色 套装', '蓝色 套装', '大红色 套装', '大红色 单件',
                '黑色 单件', '蓝色 单件', '浅紫', '紫色套装（其他颜色备注）',
                '粉色套装（含内裤）', '虾粉'], dtype=object)
```

那么一点布料，居然搞得这么复杂。

```
In [3]: pd.DataFrame(bras['productColor'].unique()).to_csv(
                    '/home/qiwsir/Documents/DataAnalysis/chapter04/colors.csv')
```

把那些颜色分类保存到一个 CSV 文件中，然后将那么多复杂的颜色分别映射为几种简单颜色（这里没有编写程序，而是手动修改 CSV 文件），于是结果如下。

```
In [4]: color_map = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter04/
                                colors.csv")
        colors = pd.DataFrame({'productColor':color_map.values[0:,1], 'color':color_
                               map.values[0:,2]})
        colors.head()
Out[4]:     color   productColor
        0    棕色        22 咖啡色
        1    粉色          02 粉色
        2    蓝色         071 蓝色
        3    黑色         071 黑色
        4    肤色         071 肤色
```

然后将 colors 与 bras 合并，即增加一个“color”列。

```
In [5]: cbras = pd.merge(bras, colors, on="productColor", how="left")
        cbras.head()
Out[5]:              creationTime    productColor    productSize     color
        0     2016-06-08 17:17:00        22 咖啡色            75C      棕色
        1     2017-04-07 19:34:25        22 咖啡色            80B      棕色
        2     2016-06-18 19:44:56          02 粉色            80C      粉色
        3     2017-08-03 20:39:18        22 咖啡色            80B      棕色
        4     2016-07-06 14:02:08        22 咖啡色            75B      棕色
```

接下来要做的就是分组统计“color”列的数据，然后以柱状图显示统计结果，如图 4-2-1 所示。

```
In [6]: color_count = cbras.groupby('color').count()    #①
        datas = color_count['productColor']
        labels = datas.index
        position = range(len(datas.index))

        from matplotlib.font_manager import *     #②
        myfont = FontProperties(fname='/usr/local/lib/python3.5/dist-packages/
                 matplotlib/mpl-data/fonts/ ttf/MSYH.TTC')
        matplotlib.rcParams['axes.unicode_minus']=False

        plt.bar(left=position, height=datas.values, width=0.6, tick_label=labels)
        plt.xticks( position, labels, fontproperties=myfont)
```

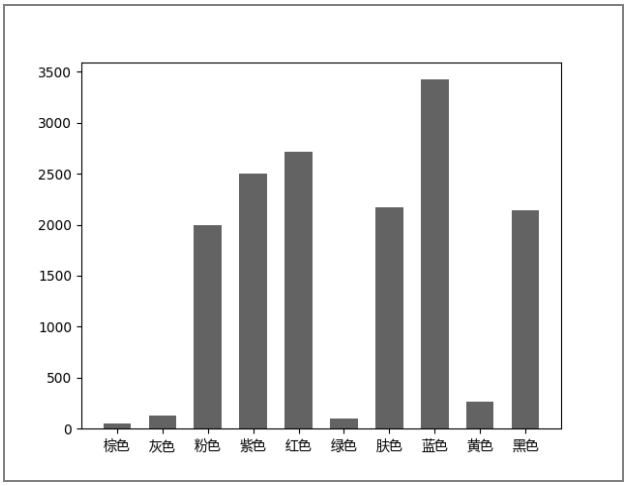


图 4-2-1　各种颜色的统计

In[6]中的①实现了分组统计，然后将统计结果在柱状图中显示。这里增加②及后续两条语句，目的是要实现在统计图中显示中文。

从统计结果中，可以看出女士们喜欢的颜色。

颜色仅仅是一个维度，还有一个更重要的维度——看数据，勿联想。

```
In [7]: bras2 = bras['productSize'].str.upper()
        bras2.unique()
Out[7]: array(['75C', '80B', '80C', '75B', '70C', '85B', '70B', '85C', '75C/34C',
               '80B/36B', '85C/38C', '85A/38A', '85B/38B', '80A/36A', '70A/32A',
               '80C/36C', '75B/34B', '75A/34A', '70B/32B', '70C/32C', 'B80', 'B75',
               'C80', '170/82/XL', 'C75', '160/70/M', 'B70', '165/76/L', 'C70',
               nan, '90C/40C', '90B/40B', '85D/38D', '85B+(内裤)套装', '85E/38E',
               '80D/36D', '90D/40D', '80E/36E', '75E/34E', '90E/40E', '75D/34D',
               '95C', '95E', '85E+(内裤)套装', '95D', '75B+(内裤)套装', '75B=34B',
               '80B=36B', '80C=36C', '90D=40D', '85B=38B', '80A=36A', '85C=38C',
               '90B=40B', '75A=34A', '90C=40C', '85A=38A', '75C=34C', '85/38C',
               '75B/34', '85B/38', '80B/36', '70B/32', 'A75', 'A80', 'A70', '75A',
               '80A', '70A', '85A', '70A=32A', '70B=32B', 'A85', 'C85', 'B85',
               '90C', '40/90A=XL 码', '34/75D=L 码', '32/70B=S 码', '36/80B=L 码',
               '38/85A=L 码', '38/85C=XL 码', '36/80C=L 码', '38/85B=XL 码',
               '38/85D=XL 码','34/75B=M 码', '34/75C=M 码', '34/75A=S 码',
               '40/90C=XL 码', '36/80A=M 码','75B=34B ', '34/75AB 中厚 2CM', '75B=34AB',
               '80B=36AB', '75B  ','38/85AB 中厚 2CM', '34/75C 薄款 0.8CM', '80B ',
               '85B=38AB  ', '85B=38AB','70B=32AB', '80A=36A 厚杯', '70A=32A 厚杯',
               '75A=34A 厚杯','75B=34B（粉色预发货 17 号）', '75B=34B 粉色预计 4 天发',
               '75B=34B（粉色预发货 20 号）', '75B=34B（粉色预发货 26 号）', '34B/75B',
               '34/75B', '40C/90C', '32B/70B', '34A/75A', '36C/80C', '34C/75C',
               '36B/80B', '34B=75B', '36A/80A', '32A/70A', '38B/85B', '38A/85A'],
              dtype=object)
```

与前面的处理流程一样，要从大小的角度对数据进行“清洗”，然后分组统计，最后绘图显示，如图 4-2-2 所示。

```
In [8]: cup = bras2.str.findall("[a-zA-Z]+").str[0]    #用正则表达式进行初步清洗
        cup2 = cup.str.replace('M', 'B')    #将其他型号归类为 A～E
        cup3 = cup2.str.replace('L', 'C')
        cup4 = cup3.str.replace('XC', 'D')
        cup5 = cup4.str.replace('AB', 'B')
        bras['cup'] = cup5

        cup_count = bras.groupby('cup').count()    #分组统计数量

        labels = ['A', 'B', 'C', 'D', 'E']
        fig, ax = plt.subplots()
        explode = (0, 0.1, 0, 0, 0)
        ax.pie(cup_count['productColor'], explode=explode, labels=labels,
               autopct='%1.1f%%', radius=1.2, startangle=0)    #绘制饼图
        ax.set(aspect='equal')
```

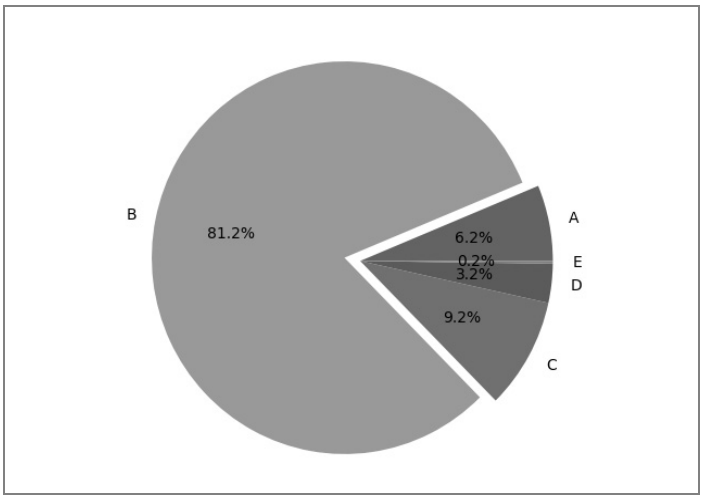


图 4-2-2　不同尺寸所占比例统计图

数据不骗人。

也不一定，关键看分析者怎么表达——归根结底还是人在玩花样。

4.3　分析电影票房数据

每年 5 月到 10 月期间，家门口的邻里中心广场上，每周都有几个晚上放露天电影，虽然不是电影院里面的大片，但也能吸引一些人围观——当然是免费的了。

电影，真神奇。

它的票房，更神奇。我们来看个究竟。

先安装获取数据的模块——Tushare（http://tushare.org），此模块的一个依赖是 lxml。

```
$ sudo pip3 install lxml
$ sudo pip3 install Tushare
In [1]: import tushare
```

```
        print(tushare.__version__)
Out[1]: 1.0.5
```

如果读者所安装的版本与此处有差异，请保持镇定，一般不会有多大影响。

```
In [2]: import tushare as ts
        df = ts.month_boxoffice('2017-8')
        df
```

Out[2]:（输出结果如图 4-3-1 所示）

	Irank	MovieName	WomIndex	avgboxoffice	avgshowcount	box_pro	boxoffice	days	releaseTime
0	1	战狼2	8.53	36	39	57.6	424048	31	2017-07-27
1	2	三生三世十里桃花	6.53	35	24	7.3	53425	29	2017-08-03
2	3	杀破狼·贪狼	7.71	32	20	6.7	49095	15	2017-08-17
3	4	星际特工:千星之城	7.45	35	19	4.6	34152	7	2017-08-25
4	5	心理罪	7.23	33	20	3.9	29015	21	2017-08-11
5	6	侠盗联盟	6.52	31	21	3.2	23655	21	2017-08-11
6	7	二十二	8.72	30	18	2.3	16830	18	2017-08-14
7	8	建军大业	8.35	33	17	2.1	15807	31	2017-07-27
8	9	十万个冷笑话2	7.85	31	15	1.7	12589	14	2017-08-18
9	10	鲛珠传	6.47	35	14	1.5	11341	21	2017-08-11
10	999	其他		33	12	9.1	66715		

图 4-3-1　输出结果

电影的暑期档历来受重视，因此只研究 2017 年 8 月的票房数据。图 4-3-1 中各列的含义依次如下。

- Irank：排名。
- MovieName：电影名称。
- WomIndex：口碑指数。
- avgboxoffice：平均票价。
- avgshowcount：场均人次。
- box_pro：月度占比。
- boxoffice：单月票房（万元）。
- days：月内天数。
- releaseTime：上映日期。

做一个简单图示，看看票房的分布，如图 4-3-2 所示。

```
In [3]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        colors = ['red', 'yellow', 'blue']
        pie = plt.pie(df['boxoffice'], labels=df['MovieName'], autopct='%1.1f%%',
                      startangle=0, colors=colors)
        plt.axis('equal')

        from matplotlib.font_manager import *      #①
```

```
        myfont = FontProperties(fname='/usr/local/lib/python3.5/dist-packages/
                 matplotlib/mpl-data/fonts/ ttf/MSYH.TTC')
                 for font in pie[1]:
                     font.set_fontproperties(myfont)
```

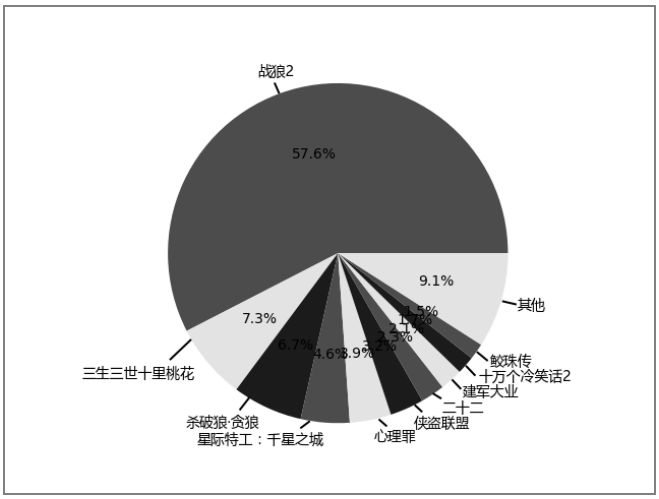


图 4-3-2　月度票房统计

In[3]中简单统计了 2017 年 8 月这个暑期档中单月电影票房，从饼图中可以清晰地看出，有一部影片票房收入居然多过了一半，真是“赢者通吃”。

注意，In[3]中①及其后面的语句，解决图 4-3-2 中的汉字显示问题。

下面就对若干年暑期档（8 月）的票房收入做一个梳理，看看能不能找到什么规律。

```
In [4]: box_office = pd.DataFrame()
        for y in range(2010, 2018):
            ym = str(y) + '-8'
            df = ts.month_boxoffice(ym)
            box_office[ym] = df['boxoffice']
        box_office
```

Out[4]:

	2010-8	2011-8	2012-8	2013-8	2014-8	2015-8	2016-8	2017-8
0	26918	39162	27993	63665	39313	73427	98811	424048
1	17760	30890	23051	29567	36338	54977	58144	53425
2	8816	22919	15020	29460	28981	48912	38770	49095
3	8125	17219	14636	17121	20807	20948	38675	34152
4	5157	6243	13518	16468	18746	19542	34589	29015
5	5133	5326	8895	14473	18714	18083	26931	23655
6	3980	3594	7037	11416	16926	17220	16663	16830
7	2655	3410	6868	7542	9770	14643	15334	15807
8	2581	3301	4159	6286	9729	14458	14736	12589
9	2233	1954	3471	5440	9464	9977	8376	11341
10	8245	13137	17216	24674	53426	69576	54576	66715

如此，得到了从 2010 年 8 月—2017 年 8 月暑期档的票房相关数据。注意，上述 DataFrame 中的数字都是字符串，需要转换为整数，然后计算当月票房总和。

```
In [5]: box_office = box_office.astype(np.int)
```

```
        total = box_office.sum()
        total
Out[5]: 2010-8    91603
        2011-8   147155
        2012-8   141864
        2013-8   226112
        2014-8   262214
        2015-8   361763
        2016-8   405605
        2017-8   736672
        dtype: int64
```

看数据表，得到的信息有限，还是要绘图，如图 4-3-3 所示。

```
In [6]: date_index = pd.to_datetime(total.index)    #将索引格式转换为日期 leixing
        fig, ax = plt.subplots()
        ax.grid(color='gray')
        ax.set_xlim((1, 10))    #设置 X 轴的范围
        ax.set_ylim((91603, 750000))    #设置 Y 轴的范围
        ax.plot(range(1, 9), total.values, marker='o', markerfacecolor='r')#绘制折线图
        ax.set_xticklabels(date_index.year)    #设置 X 轴主刻线的标示
        ax.set_title("Box Office of August")    #设置图示标题
```

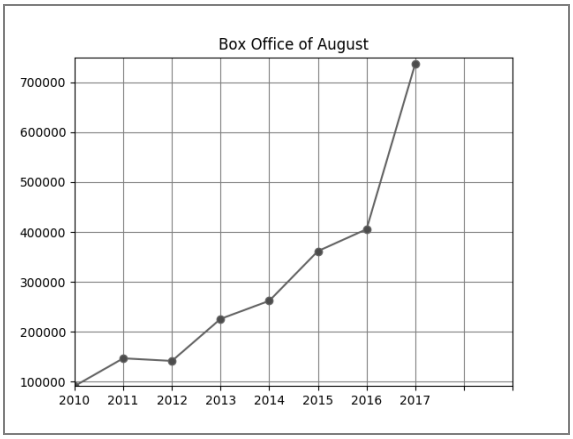


图 4-3-3　历年暑期档票房收入折线图

观察图 4-3-3 的曲线特点，不觉得 2017 年那个数据有点反常吗？

所谓“常”，就是按照 2010 年—2016 年的数据走势——这里又遇到了“回归”，可以判断后面数据的大体范围，而实际上 2017 年的数据没有依照上述之“常”，故曰“反常”。

依据“回归”思想，判断一下 2018 年 8 月的票房收入——60 亿元上下，不会创造 2017 年的纪录——这是“常”，试看会不会被“反”。

下面试着探究一下 2017 年数据暴涨的原因。

```
In [7]: first = box_office.loc[0].astype(np.int)    #历年票房第一
        after_first = total - first    #其他影片票房收入

        date_index = pd.to_datetime(total.index)
```

```
fig, ax = plt.subplots()
ax.grid(color='gray')
ax.bar(range(1, 9), after_first.values, label='others')    #①
ax.bar(range(1, 9), first.values, bottom=after_first.values, label='first')
#②
ax.set_xticklabels([0]+date_index.year.tolist())
plt.legend(loc=0)
```

①和②绘制两个柱状图，以 after_first 为底，以 first 为顶，两者结合就是当年暑期档（8 月）票房总收入，如图 4-3-4 所示。

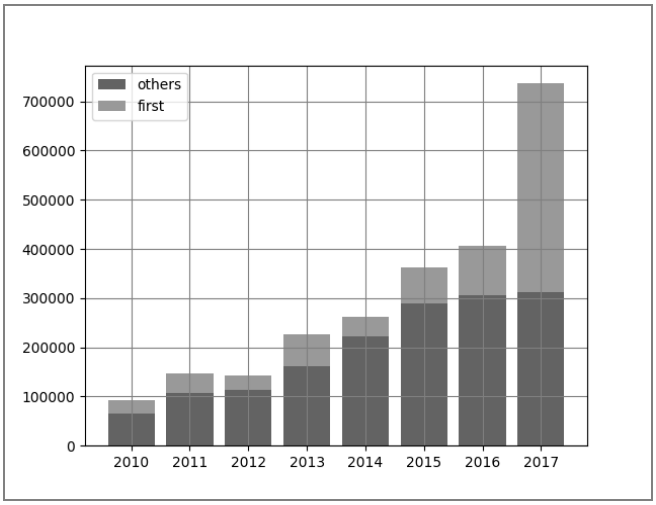


图 4-3-4　历年暑期档票房收入柱状图

从图 4-3-4 不难看出，对 2017 年票房起到拉升作用的就是那一部影片，如果除去它，票房较前几年增长不是很多。是不是 2018 年还会出现类似的“票房助推器”呢？拭目以待。否则，就“回归”了。

当然，这里不是旨在评价电影市场，而是借用相关数据，演练如何进行数据分析。

4.4　可视化城市人口数据

众多高手云集华山之巅，举办第 1024 届华山“论剑嘉年华”。

“华山”在哪里？打开搜索引擎的地图，搜索即可见。

此时，你正在使用地理信息数据。

人类很早就已经尝试把地球“数字化”了，比如规定了经度、纬度和海拔高度，就是“数字化”的典型，从此地面上的每个位置都可以精确描述了。

众多高手云集于东经 109°57′、北纬 34°25′、海拔 2154.9m 处论剑。

地理信息中的数据需要可视化，才能让受众理解。

用什么工具实现可视化？推荐 Basemap。

1．安装 Basemap

Basemap 是 Matplotlib 的一个子模块，其作用主要是绘制地图。

不过，要使用它，需要经历一个比较麻烦的安装过程。

如果访问 https://matplotlib.org/basemap/，就可以看到 Basemap 的完整文档，包括但不限于源码下载地址、安装方法、应用方法和举例。下面所述，多少有点翻译文档的味道了。

首先安装各种依赖。第一个就是 PROJ4（http://proj4.org/），以下安装流程供参考。

```
$ sudo pip3 install pyproj
```

第二个是 GEOS (Geometry Engine - Open Source)，官方网站为 https://trac.osgeo.org/geos/（或许访问有困难），找到合适的压缩包，下载后安装。

```
$ tar xvjf geos-3.6.2.tar.bz2
$ cd geos-3.6.2/
geos-3.6.2$ configure
geos-3.6.2$ make
geos-3.6.2$ sudo make install
```

还要检查一下计算机中是否安装了 Pillow，如果读者能够按部就班地跟着本书敲代码，则这个工具肯定安装过了。

再对照官方文档检查一下，确认所需的各种依赖都安装完毕。

最后就是到 https://github.com/matplotlib/basemap/releases/网站下载 Basemap 源码，然后解压缩，进入其目录中，执行一下安装命令。

```
$ sudo python3 setup.py install
```

至此，安装完毕。在 Jupyter 中执行以下语句。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from mpl_toolkits.basemap import Basemap
```

若没有报错，则说明安装成功。

另外，如果读者使用 Anaconda 管理 Python 环境，可以直接使用 conda install basemap 进行安装，各种依赖会自动被装上。

只要能安装成功，哪种方法都可以。

2．简单应用

准备工作妥当之后，先画一张简单的图。不过，本书此处不显示地图内容，只给出效果图的网址，读者若有兴趣可以去查看。

依然强调，阅读本书的最佳方法是一边看书，一边敲代码。

```
In [2]: from mpl_toolkits.basemap import Basemap    #①
        fig = plt.figure(figsize=(8, 8))
        m = Basemap(projection='lcc', resolution=None,
                    width=8E6, height=8E6, lat_0=30, lon_0=120)    #②
```

```
m.etopo(scale=0.5, alpha=0.5)    #③

x, y = m(120.7, 31.3)    #④
plt.plot(x, y, 'ok', markersize=5)    #⑤
plt.text(x, y, ' Soochow', fontsize=12)    #⑥
```

效果图网址：https://github.com/qiwsir/DataAnalysis/blob/master/chapter04/Figure_441.png。

因为是第一次使用 Basemap，所以有必要对其应用方法做较为详细的解释。

语句①引入了 Basemap 模块，Basemap()本质上是一个类，利用它可以创建一个 Basemap 实例，这就是语句②的功效了。有了 Basemap 对象，才能有地图，才能在地图上可视化地理数据。所以，有必要隆重介绍 Basemap()类。

完整的参数列表如下：

```
Basemap(llcrnrlon=None, llcrnrlat=None, urcrnrlon=None, urcrnrlat=None, llcrnrx=None,
llcrnry=None, urcrnrx=None, urcrnry=None, width=None, height=None, projection='cyl',
resolution='c',  area_thresh=None,  rsphere=6370997.0,  ellps=None,  lat_ts=None,
lat_1=None, lat_2=None, lat_0=None, lon_0=None, lon_1=None, lon_2=None, o_lon_p=None,
o_lat_p=None, k_0=None, no_rot=False, suppress_ticks=True, satellite_height=35786000,
boundinglat=None, fix_aspect=True, anchor='C', celestial=False, round=False, epsg=None,
ax=None)
```

依据经验，在 Jupyter 中可以用“Basemap?”方式查看完整的文档说明，读者有空阅读一下更好。这里只选择几个现在用得着的参数进行说明。

- projection：地球不是二维的，而画出来的地图是二维的，这就需要转换，转换方法是将原来的三维投影为二维。在实践中有多种投影方法，不同的投影方法转换的结果不同。比如谷歌地图使用的是 Mercator（“merc”）投影；②中 projection='lcc'，采用的是 Lambert Conformal 投影。projection 的默认值是“cyl”，即 Cylindrical 投影（圆柱投影）。读者可以在 Basemap 的文档中看到所有的 projection 可选值。
- width/ height：在 projection 所规定的投影模式下，所展示的地区区域的宽度/高度，以 m 为单位。
- lon_0/ lat_0：用经纬度标明地区的中心位置。lon_0 表示经度（longitude），lat_0 表示纬度（latitude）。
- resolution：用字符串表示分辨率，比如默认 resolution='c'。表 4-4-1 为所有可选值及其说明。

表 4-4-1　Basemap 实例对象分辨率可选值

可选值	说　明
c	crude，粗糙
l	low，低
i	intermediate，中等
h	high，高
f	full，完全，即分辨率最高
None	对该 Basemap 实例对象不设置分辨率

这样，②就创建了一个 Basemap 实例对象。这个实例对象同样具有不少方法和属性，比如③。

m.etopo()方法依据 Basemap 实例对象的地形数据生成图片，并作为绘图对象的背景，即得到了以创建 Basemap 实例对象所选定的地图为背景的画布。这个背景图的大小默认为 5400 × 2700 像素，参数 scale 能够调整显示比例。

完成以上工作之后，笔者打算在地图上标出“苏州”，然而需要先解决一个问题。根据以往的绘图经验，在图上有坐标系（简称“图上坐标”），而地图中往往用经纬度表示一个具体位置（简称“地理坐标”），比如苏州的地理坐标是东经 120.7°、北纬 31.3°。现在要在图上标出“苏州”的位置，用哪个坐标？

当然是“图上坐标”。

现在能够知道的是苏州的“地理坐标”值。

所以要建立“图上坐标”和“地理坐标”的对应关系——语句④的作用。

最后用⑤⑥完成对“苏州”城市位置的标注和说明。

这就是 Basemap 的简单应用。下面要在此基础上做一个复杂的东西。

3．再绘散点图

曾记否，本书前面绘制了展示江苏省各个城市的面积人口和所在位置的散点图。现在我们要在原有基础上更进一步地用地图作为图示的背景。

```
In [3]: cities = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter03/city_
                             population.csv")
        lat = cities['latd'].values
        lon = cities['longd'].values
        population = cities['population'].values
        area = cities['area'].values/10

        fig = plt.figure(figsize=(8, 8))
        m = Basemap(projection='lcc', width=5E5, height=6E5, lat_0=32.9,
                    lon_0=119.2,resolution='h')
        m.shadedrelief()    #设置地形图效果
        m.drawcoastlines(color='gray')    #描绘海岸线

        #画散点图
        m.scatter(lon, lat, latlon=True, c=np.log10(population), s=area, cmap='Reds',
                  alpha=0.5)

        plt.colorbar(label='log$_{10}$(population)')

        for area in [100, 300, 500]:
            plt.scatter([], [], c='k', alpha=0.3, s=area, label=str(area) + ' km$^2$')

        plt.legend(scatterpoints=1, frameon=False, labelspacing=1, title='City Area')
```

效果图网址：https://github.com/qiwsir/DataAnalysis/blob/master/chapter04/Figure_442.png。

在前述知识的基础上，读者应该能理解 In[3]中各行代码的含义，不过还是建议逐行进行注

释，同时在计算机上调试。

关于使用 Basemap 的更多应用举例，可以参考官方网站的有关内容。

4．Geopandas

Basemap 虽然是常用的一个模块，但是我们从来不停止造“轮子”，所以就有了 Geopandas——号称后起之秀。

按照“望文生义”的原则，Geopandas 应该跟 Pandas 有关系吧。的确如此，它继承了 Pandas 的很多使用习惯，所以本书的读者使用 Geopandas 会更得心应手。

Geopandas 的官方网站为 http://geopandas.org/。

根据官方文档中的指导，安装 Geopandas 及其各种依赖——文档中已经给出了安装方法，下面仅仅是一种安装过程的指令罗列，在本书前面的讲解中已经安装过，不再重复。

```
$ sudo pip3 install Shapely
$ sudo pip3 install fiona
$ sudo pip3 install geopy
$ sudo pip3 install psycopg2
```

打开 http://libspatialindex.github.io/网站，根据自己的操作系统下载 libspatialindex 包，然后编译/安装。以下是笔者在 Ubuntu 系统上的操作流程，供参考。

```
$ curl -L http://download.osgeo.org/libspatialindex/spatialindex-src-1.8.5.tar.gz | tar
xz
$ cd spatialindex-src-1.8.5
spatialindex-src-1.8.5$ ./configure    #如果这里指定了路径，可以省略后面的①
spatialindex-src-1.8.5$ make
spatialindex-src-1.8.5$ sudo make install
spatialindex-src-1.8.5$ sudo ldconfig    #①
```

再根据 http://toblerity.org/rtree/所提供的安装方法安装 Rtree 及其他依赖程序。

```
$ sudo pip3 install rtree
$ sudo pip3 install descartes
$ sudo pip3 install pysal
```

依赖安装好之后，执行安装 Geopandas 的命令。

```
$ sudo pip3 install geopandas
```

至此，大功告成。

如果读者使用 Anaconda 管理开发环境，则可以直接使用下面的命令：

```
conda install -c conda-forge geopandas
```

这个命令也连带安装了相应的依赖。当然，不一定一次安装成功。

配置环境要有耐心，并考验读者的搜索能力（建议使用 Google）。注意笔者提倡的搜索引擎，可能决定了读者的成败——这都是笔者趟过的“坑”。

此处不讲述 Geopandas 的基本知识。只要读者严格遵循本书指导进行学习，在正常情况下，到目前已经掌握了快速学习新东西的方法了——授之以渔，就是本书的终极目标。所以，读者能够通过阅读官方文档掌握 Geopandas 的基本知识。

下面直接看应用示例。

示例还是老的，即用 Geopandas 再次绘制江苏省各城市人口分布图。

在敲代码之前，先下载 SHP 格式的文件，在下载网站（http://www.diva-gis.org/gdata）的“Country”下拉列表框中选择“China”，在“Subject”列表框中选择“Administrative areas”，如图 4-4-1 所示。

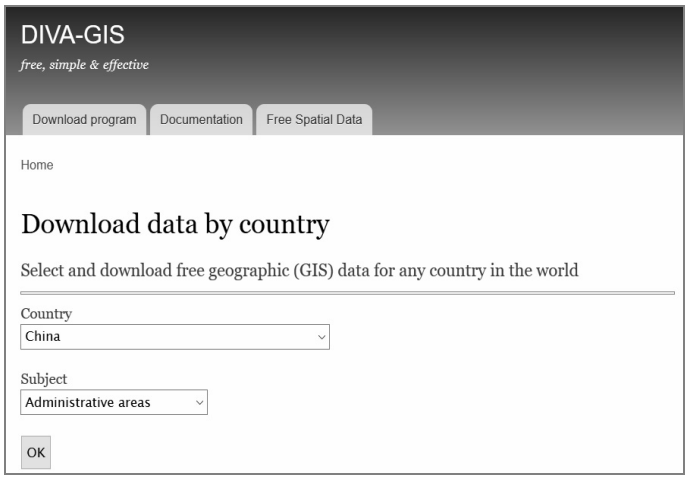


图 4-4-1　下载 SHP 格式的文件

所谓 SHP 格式的文件，就是 ESRI Shapefile（SHP）或简称 Shapefile，是美国环境系统研究所公司（ESRI）开发的一种空间数据开放格式。目前，该文件格式已经成为地理信息软件界的一个开放标准。如果读者从事这方面的业务，可以进行深入研究。

文件下载后，解压缩到某处，我们会选择使用里面的某些文件。

```
In [4]: import geopandas as gp
        #从 SHP 文件中读取中国政区地理数据
        cn_shp = gp.read_file(
                "/home/qiwsir/Documents/data_analysis/chapter04/CHN_adm/CHN_adm2.shp")
        #把江苏省的地理政区数据单独取出
        js_shp = cn_shp[cn_shp.NAME_1=="Jiangsu"]
        js_geo = gp.GeoDataFrame(js_shp)    #转换为 Geopandas 的数据格式

        cities = pd.read_csv("/home/qiwsir/Documents/data_analysis/chapter03/city_
                             population.csv")
        js_data = gp.GeoDataFrame(cities)    #转换为 Geopandas 的数据格式

        #将 js_geo 中的 NL_NAME_2 列名称修改为 name
        js_geo = js_geo.rename(index = str, columns={"NL_NAME_2":"name"})

        #根据 name 列合并 js_geo 和 js_data 的数据
        jiangsu = js_geo.merge(js_data, on='name', how='left')

        #绘制人口分布图
        jiangsu.plot('population', k=4, cmap=plt.cm.Greens, alpha=1,legend = True)
```

```
    plt.gca().xaxis.set_major_locator(plt.NullLocator())    #去掉 X 轴刻度
    plt.gca().yaxis.set_major_locator(plt.NullLocator())    #去年 Y 轴刻度
    for row in jiangsu.itertuples():    #标记各城市名称
        plt.text(row.geometry.centroid.x, row.geometry.centroid.y, row.NAME_2)
```

上述程序运行的效果图网址是 https://github.com/qiwsir/DataAnalysis/blob/master/chapter04/Figure_444.png。

读者不妨比较一下 Geopandas 和 Basemap 两个库，是否觉得“轮子”还是在不断发展进化的？或许各有优势，那就根据自己的喜好和项目要求选择吧。

关于地理信息数据的分析处理和可视化，是一个专业方向，此处示例仅仅是带领读者观其大略，连入门都谈不上。如果读者有意愿在此领域深入探索，应该寻找更专业的书籍和资料研习。

4.5　分析希腊葡萄酒数据

本节应用示例来自于网友 Florents 的工作，笔者在编写本书的时候，得到了他通过电子邮件的授权，可以将其发表的文章 *Analyzing 1000+ Greek Wines With Python* 中的数据分析过程引用到本书（网址：https://github.com/Florents-Tselai/greek-wines-analysis/blob/master/greek-wines-analysis.ipynb）。

为了适合本书的读者，笔者在引用 Florents 的工作成果的过程中，做了适当修改。

首先感谢 Florents（Thanks, Florents）——虽然他不懂中文，也可能看不到本书。

“葡萄美酒夜光杯，欲饮琵琶马上催。醉卧沙场君莫笑，古来征战几人回？”

唐朝人已经在喝葡萄酒了。现在，我们津津乐道的总是法国葡萄酒——“来瓶 82 年的拉菲”。

Florents 或许是“酒中仙”，不过在他的文章中分析的是希腊的葡萄酒数据。当然，我们的重点不在于什么酒，而在于数据及其分析方法。

分析数据的前提是要先搞到数据，搞到数据的一种常用方法就是利用网络爬虫技术从网页上获取，在 4.2 节曾用过。而这次，Florents 提供了一个爬虫工具，还把这个工具做成了可安装的第三方模块，放在了 https://github.com/Florents-Tselai/greek-wines-analysis 上，供读者免费下载——一定要下载，因为他已经把要分析的数据集整理好，公开分享，不用读者自己到网站上爬取了。

从上述网址下载代码，编译安装 Florents 提供的工具。

```
$ git clone https://github.com/Florents-Tselai/greek-wines-analysis.git
$ cd greek-wines-analysis/
greek-wines-analysis$ sudo python3 setup.py install
```

安装过程中会检测有关依赖，缺少的会自动安装。

安装完毕，体验一下如何使用此工具获取数据。

```
In [1]: from houseofwine_gr import get
        get('http://www.houseofwine.gr/how/megas-oinos-skoura.html')
Out[1]: {'ageable': True,
```

```
        'alcohol_%': 13.5,
        'avg_rating_%': 82,
        'color': 'Ερυθρός',
        'description': 'Στιβαρό ερυθρό κρασί παλαίωσης. Βασίζεται σε χαρμάνι που
        παντρεύει το πικάντικο σκέρτσο του Αγιωργίτικου με την αυστηρή δύναμη του Cabernet
        Sauvignon, στη ζεστή και βανιλάτη αγκαλιά του δρύινου βαρελιού που τα φιλοξένησε
        κατά την 20μηνη παλαίωσή του. Ποιοτική ετικέτα για μεγάλα φαγοπότια ή εκλεκτά
        κελάρια.',
        'drink_now': False,
        'keep_2_3_years': False,
        'n_votes': 22,
        'name': 'Μέγας Οίνος Σκούρα 2014',
        'price': 20.9,
        'tags': ['Αγιωργίτικο', 'Cabernet Sauvignon', 'Ξηρός', 'Ήπιος'],
        'url': 'http://www.houseofwine.gr/how/megas-oinos-skoura.html',
        'year': 2014}
```

的确是简单好用的工具。

不过，真正爬取数据的过程，还是留给读者去研究，笔者在这里直接使用 Florents 提供的数据集。

```
In [2]: %matplotlib inline
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import matplotlib as mpl
        import seaborn as sns

        plt.style.use('fivethirtyeight')    #①

        mpl.rcParams['figure.figsize'] = (8,6)
        mpl.rcParams['font.size'] = 14
        mpl.rcParams['font.family'] = 'Serif'
        mpl.rcParams['figure.facecolor'] = 'white'
        plt.rcParams['axes.facecolor']='white'
        plt.rcParams['axes.grid'] =False
        plt.rcParams['figure.facecolor']='white'    #②

        df = pd.read_json('/home/qiwsir/Documents/DataAanalysis/chapter04/greek-
                    wines- analysis/data/houseofwine.gr-wines.json', encoding=
                    'utf-8')  #③
        df.head()
Out[2]:（输出结果如图 4-5-1 所示）
```

In[2]中①到②之间的几条语句的作用是设置本项目的显示风格，包括画布、字号、字体、网格显示与否、各部分颜色等。最后在③句中读入了 Florents 提供的数据集，请读者注意③中演示的是笔者保存数据集文件的路径，如果读者的路径有变化，在调试的时候要修改。

	ageable	alcohol_%	avg_rating_%	color	description	drink_now	keep_2_3_years	n_votes	name	price	tags	url
0	False	8.5		Λευκός	Κάποια από τα Riesling του Rheingau της Γερμαν...	False	True		Riesling Spatlese 2013	33.7	[Riesling, Ήπιος, Ημίγλυκος]	http://www.houseofwine.gr/how/riesling-spatles...
1	False	13.5		Λευκός	Τυπικό Νέο-Ζηλανδέζικο Sauvignon Blanc από ένα...	True	False		Spy Valley - Sauvignon Blanc 2016	18.3	[Ήπιος, Sauvignon Blanc, Ξηρός]	http://www.houseofwine.gr/how/spy-valley-sauvi...
2	True	13.0		Ερυθρός	Η κορωνίδα του Κτήματος Χρυσοχόου και ταυτόχρο...	False	False		Cava Χρυσοχόου 2008	25.0	[Ήπιος, Ξηρός, Ξινόμαυρο, Merlot]	http://www.houseofwine.gr/how/cava-hrysohoou.html
3	False	12.5	90	Λευκός	Ένα από τα δημοφιλέστερα λευκά κρασιά στην Ελλ...	True	False	27	Κτήμα Γεροβασιλείου - Λευκός 2016	12.1	[Ήπιος, Ξηρός, Ασύρτικο, Μαλαγουζιά]	http://www.houseofwine.gr/how/ktima-gerovasile...
4	False	13.5		Ερυθρός	Εισαγωγή στα κρασιά από τη γηγενή ποικιλία της...	False	True		Landskroon - Pinotage 2013	13.1	[Ήπιος, Ξηρός, Pinotage]	http://www.houseofwine.gr/how/landskroon-pinot...

图 4-5-1　输出结果

读入数据集之后，根据经验，要对其进行“清洗”——数据清洗是数据分析的必备工作，也是技术和艺术的结合。如果在网上搜索“数据清洗”，可以看到很多人提供的各种经验。但是，那些都是他们的。当你自己遇到了具体的业务时，他们的经验仅供借鉴，还需要“实事求是，具体问题具体分析”。

且看 Florents 是如何清洗数据的。

数据集中有不少“空”，都用 np.nan 替代，这样做之后，在运算中就不会因为“空”而报错了。

```
In [3]: df = df.replace('', np.nan, regex=True)
        df = df.rename(columns={'alcohol_%': 'alcohol', 'avg_rating_%': 'avg_rating'},
                       inplace=False)
```

在数据集中，有不少数字，因为它们是从网页上获取并保存的，在数据集中都是字符串类型，这不便于进行数学运算，所以要将其修改为浮点数或者整数类型。

```
In [4]: df['alcohol'] = df.alcohol.astype(float)
        df['n_votes'] = df.n_votes.astype(int, errors='ignore')
        df['price'] = df.price.astype(float)
        df['year'] = df.year.astype(int, errors='ignore')
```

笔者的希腊文水平停留在几个希腊字母上，还是物理老师教的，不知读者是否熟悉希腊文。倒是 Florents 很照顾我，他把数据集中的某些内容换成了英文，比如颜色（数据集的 color 列数据）。

```
In [5]: df['color'] = df.color.replace({'Λευκός': 'White', 'Ερυθρός': 'Red', 'Ροζέ':
                                        'Rosé'})
```

根据笔者所查阅的资料显示，葡萄酒可以有多种分类方式，例如下面的分类。

- 按照颜色分：红葡萄酒（Red）、白葡萄酒（White）和粉红葡萄酒（Rose）。
- 按糖度分：干型葡萄酒、半干葡萄酒、半甜葡萄酒和甜型葡萄酒。
- 按是否含二氧化碳分：静止葡萄酒、气泡葡萄酒。

经过上述操作之后，我们就可以统计不同颜色的葡萄酒的数量了，如图 4-5-2 所示。

```
In [6]: ax = df['color'].value_counts().plot('bar')
        ax.set(ylabel='Wines', title='Wine Color Frequency')
Out[6]: [<matplotlib.text.Text at 0x7fbd46e95c88>,
         <matplotlib.text.Text at 0x7fbd46ea98d0>]
```

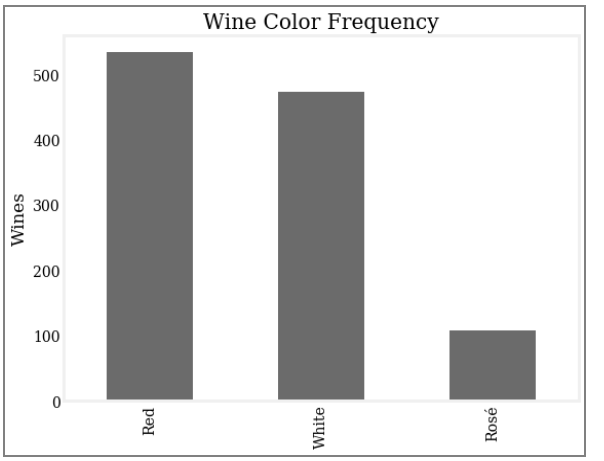


图 4-5-2　酒的颜色统计

从图 4-5-2 中不难看出，在所获得的数据集中，红葡萄酒数量最多，相比而言粉红葡萄酒就太少了——还是要真正的“红”，“粉红”毕竟不正宗。

在 In[4]的代码中，我们修改了一些列的数据类型。通过这些列的数据，我们能对该网站所售的葡萄酒有一个基本了解——开始为所售葡萄酒群体画像了，如图 4-5-3 所示。

```
In [7]: fig, ((ax1, ax2),(ax3, ax4)) = plt.subplots(ncols=2, nrows=2, figsize=(16,12))

        df.year.dropna().astype(int).value_counts().plot('bar', ax=ax1)
        ax1.set(title='Production Year Frequency', xlabel='Year');

        sns.distplot(df[df.alcohol < 100].alcohol.dropna(), ax=ax2)    #①
        ax2.set(xlabel='Alcohol %', title='Alcohol % Distribution');

        sns.distplot(df[df.price < 100].price.dropna(), ax=ax3)    #②
        ax3.set(xlabel='Price (< 100)')

        sns.distplot(df.avg_rating.dropna(), ax=ax4)
        ax4.set(xlabel='Average Rating')
```

In[7]的①和②两句中都有限制条件，之所以如此，是因为超出条件的数据太少，统计图显示的是大多数数据的统计结果。

以下是笔者对统计图的解读，仅供读者“娱乐”。

- 从年份的图示中，可以非常明显地看到，该网站所售的葡萄酒还是以近几年生产的为主。
- 从酒精含量统计分布图中可知，大多数葡萄酒的酒精含量是 10%～15%，其中还有一些是 5%，《维基百科》的“葡萄酒”词条也认为“一般葡萄酒的酒精含量为 8%～15%”。

- 在价格上，多数在 20 元左右（单位不是人民币，是欧元）。
- 在酒的平均分数统计图中，显示所售葡萄酒的评分还是蛮高的。

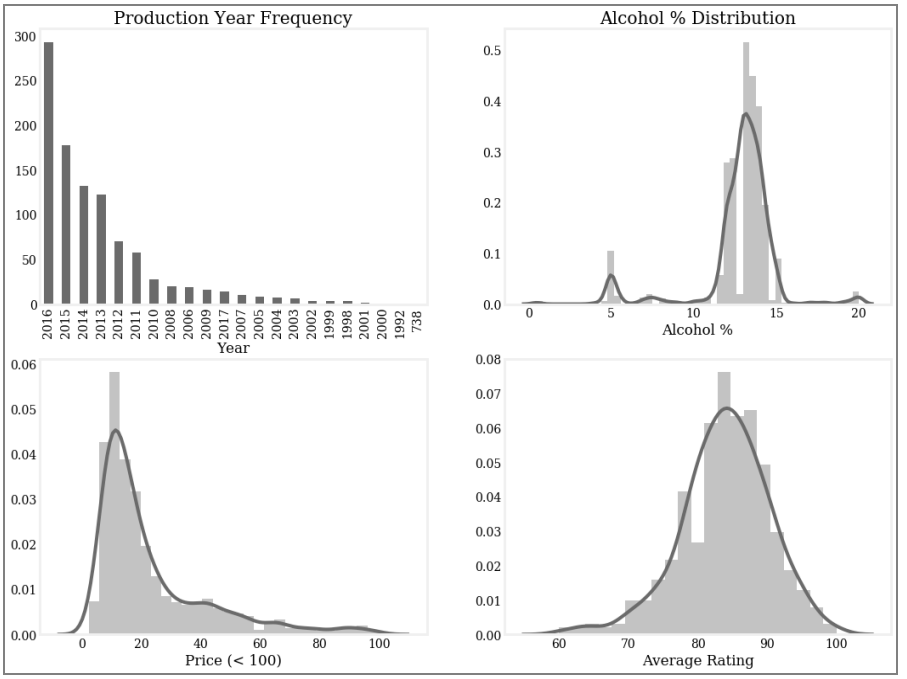


图 4-5-3　葡萄酒的不同属性

比较“价格”和“平均分数”两张图，不禁惊叹于所售葡萄酒的物美价廉——是否要感叹希腊人真厚道，顺便再发泄一下对身边商家的不满呢？

Florents 怕读者激动，在原文中补充了一个说明。原来典型的葡萄酒评级分数是 50～100，而不是 0～100。此外，一般认为 90 分以上的葡萄酒为好酒，也更好卖——这个网站好酒不多哦，不过好酒肯定贵，超出大众消费——商家都是按照市场行情来的。

其实，笔者还关心 Florents 没有关注的一个问题：有没有拉菲？

```
In [8]: wine_name = df['name']
        laf_name = []
        for name in wine_name:
            try:
                name_lst = name.split()
                if "Lafite" in name_lst:
                    laf_name.append(name)
                    #laf = df[df['name']==name]
            except:
                pass
        print(laf_name)
        ['Château Lafite Rothschild 2005']    #print()的结果
In [9]: laf = df[df['name']==laf_name[0]]
        laf[['name', 'price', 'year', 'color', 'avg_rating', 'alcohol']]
Out[9]:                          name       price    year  color  avg_rating  alcohol
        16 Château Lafite Rothschild  2005 3100.0  2005.0    Red         NaN     12.5
```

有拉菲，是 2005 年生产的，并且价格为 3100 欧元。

还是继续分析分布较为集中的数据，力争再搞点别的信息。

在 Out[2]的结果中，有 tags 一列。

```
In [10]: df.tags.head(10)
Out[10]: 0                          [Riesling, Ήπιος, Ημίγλυκος]
         1                       [Ήπιος, Sauvignon Blanc, Ξηρός]
         2                    [Ήπιος, Ξηρός, Ξινόμαυρο, Merlot]
         3                 [Ήπιος, Ξηρός, Ασύρτικο, Μαλαγουζιά]
         4                           [Ήπιος, Ξηρός, Pinotage]
         5                        [Ήπιος, Sangiovese, Ξηρός]
         6                         [Ήπιος, Ξηρός, Ξινόμαυρο]
         7    [Cabernet Sauvignon, Ήπιος, 13, Ξηρός, Merlot,...
         8                                   [Ήπιος, Γλυκός]
         9                              [Ήπιος, Αηδάνι, Ξηρός]
         Name: tags, dtype: object
```

完全看不懂输出结果，也不用看懂。据 Florents 先生说，tags 中是对每种酒特征的描述，包括酒的糖度、酿造方式等。

既然如此，就先把“糖度”的标签选出来。

```
In [11]: df['tags'] = df.tags.map(set)    #①
         sweetness_values = {'Γλυκός', 'Ημίγλυκος', 'Ξηρός', 'Ημίξηρος'}    #②

         df['sweetness'] = df.tags.map(
                                    sweetness_values.intersection
                                    ).map(lambda x: x.pop() if x else None)    #③

         translations = {'Γλυκός': 'Sweet', 'Ημίγλυκος': 'Semi-Sweet',
                         'Ξηρός': 'Dry', 'Ημίξηρος': 'Semi-Dry'}
         df['sweetness'] = df['sweetness'].replace(translations)    #④
```

正如前面所说，如果按照糖度对葡萄酒进行分类，可以分为四种，如 In[11]的②所示。语句③创建新的列 sweetness，用于保存对该葡萄酒糖度的描述。

在 In[11]的①中，将 tags 中的值转换为集合，目的在于用③中的 df.tags.map（sweetness_values.intersection）计算每种酒的 tags 值与 sweetness_values 的交集，这样就不用判断 tags 的值中是否有描写糖度的词语，也不用捕获错误；再从交集中取出描述糖度的字符串，即语句③中的 map(lambda x: x.pop() if x else None)的作用。

语句④的作用依然是将希腊文替换为英文。

这样，就在数据集中增加了一列 sweetness，用于保存葡萄酒糖度。

葡萄酒还可以按是否有气泡划分。起泡是因为含有二氧化碳，类似于碳酸饮料。在 tags 中也有标签描述这个特征。

```
In [12]: df['sparkling'] = df.tags.map({'Αφρώδης', 'Ημιαφρώδης'}.intersection
                                ).map(lambda x: x.pop() if x else None
                                ).replace({'Αφρώδης': 'Sparkling', 'Ημιαφρώδης':
                                'Semi-Sparkling'}
```

```
                        )
    df['sparkling'] = df.sparkling.fillna('Not Sparkling')
```

In[12]的方法与 In[11]一样，不再赘述。

在 Florents 的操作中，还有 is_mild 列，猜测是用来描述口感的吧——仅仅是猜测。

```
In [13]: df['is_mild'] = df.tags.map(lambda x: 'Ήπιος' in x)
```

综上，就在数据集中增加了 sweetness、sparkling、is_mild 三列，再加上原有的 color 列，分别表征了葡萄酒的不同特征属性。下面作图，分别从上述四个维度进行统计，如图 4-5-4 所示。

```
In [14]: fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2, sharex=True,
                                                      squeeze=False)

    for attr, ax in zip(['sweetness', 'color', 'sparkling', 'is_mild'], (ax1, ax2,
                        ax3, ax4)):
        df[attr].value_counts().sort_values(ascending=True).plot(kind='barh',ax=ax)
        attr_str = 'Mildness' if attr is 'is_mild' else attr.title()
        ax.set(xlabel='Number of Wines', title='{} Frequency'.format(attr_str));

    fig.set_size_inches((12,8))
    fig.tight_layout()
```

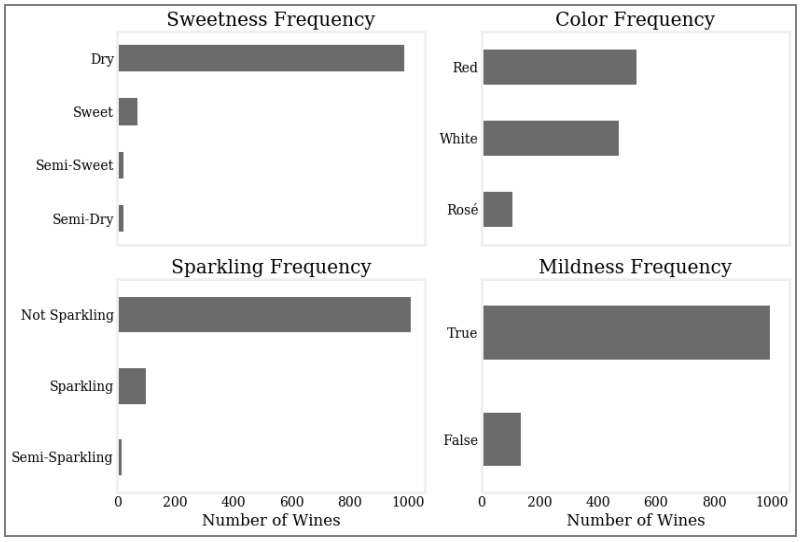


图 4-5-4　统计葡萄酒的不同特征

据笔者查阅的关于葡萄酒的资料显示，每种酒的最重要的特色取决于葡萄。用不同的葡萄酿制的葡萄酒会有所差别。接下来就研究数据集中酿制葡萄酒的葡萄种类。

经过前述的操作之后，可以放心大胆地把 tags 中与 sweetness、sparkling、 is_mild 三列值重复的标签去除。

```
In [15]: non_varietal_tags =
             {'Αφρώδης', 'Ημιαφρώδης', 'Ξηρός', 'Ημίξηρος', 'Γλυκός', 'Ημίγλυκος',
              'Ήπιος'}
         df['varieties'] = df.tags.map(lambda t: t.difference(non_varietal_tags)) #①
```

实际上 tags 的值没有进行修改，这也是数据清洗的习惯，尽可能不轻易修改原始数据。

In[15]中的语句①依然采用集合运算的方法，在数据集中建立 varieties 列，保存酿制葡萄酒的葡萄品种名称——是否可以认为 tags 中除 non_varietal_tags 值外，其他字符串都是葡萄品种名称呢？Florents 认为可以，我们姑且也这么认为吧。

仔细查看 df['varieties']的数据，里面居然包含了整数，显然这不是我们需要的。

```
In [16]: def is_not_int(x):
             try:
                 int(x)
                 return False
             except ValueError:
                 return True

         df['varieties'] = df.varieties.map(lambda x: set(filter(is_not_int, x)))
```

清洗了 varieties 的数据之后，再增加一列 is_varietal，用以标记该葡萄酒是用单一品种的葡萄酿制而成的，还是混合使用了多种葡萄。

```
In [17]: df['is_varietal'] = df.varieties.map(set.__len__) == 1
         df[['name', 'tags', 'is_varietal', 'varieties']].head()
```

Out[17]:（输出结果如图 4-5-5 所示）

	name	tags	is_varietal	varieties
0	Riesling Spatlese 2013	{Ήπιος, Ημίγλυκος, Riesling}	True	{Riesling}
1	Spy Valley - Sauvignon Blanc 2016	{Ξηρός, Ήπιος, Sauvignon Blanc}	True	{Sauvignon Blanc}
2	Cava Χρυσοχόου 2008	{Ξηρός, Ήπιος, Merlot, Ξινόμαυρο}	False	{Merlot, Ξινόμαυρο}
3	Κτήμα Γεροβασιλείου - Λευκός 2016	{Ξηρός, Ήπιος, Μαλαγουζιά, Ασύρτικο}	False	{Ασύρτικο, Μαλαγουζιά}
4	Landskroon - Pinotage 2013	{Ξηρός, Ήπιος, Pinotage}	True	{Pinotage}

图 4-5-5　输出结果

如果 is_varietal 的值为 True，则很有必要把该葡萄的名称单独列出，于是再创建 single_variety 列，记录符合此条件的葡萄名称，否则记录为 NaN。

```
In [18]: df.loc[df.is_varietal, 'single_variety'] =
                     df.loc[df.is_varietal, 'varieties'].map(lambda v: next(iter(v)))
         df[['name', 'is_varietal', 'varieties', 'single_variety']].head()
```

Out[18]:（输出结果如图 4-5-6 所示）

	name	is_varietal	varieties	single_variety
0	Riesling Spatlese 2013	True	{Riesling}	Riesling
1	Spy Valley - Sauvignon Blanc 2016	True	{Sauvignon Blanc}	Sauvignon Blanc
2	Cava Χρυσοχόου 2008	False	{Merlot, Ξινόμαυρο}	NaN
3	Κτήμα Γεροβασιλείου - Λευκός 2016	False	{Ασύρτικο, Μαλαγουζιά}	NaN
4	Landskroon - Pinotage 2013	True	{Pinotage}	Pinotage

图 4-5-6　输出结果

用同样的操作方法，增加标记是否为多种葡萄酿制的列 is_blend。

```
In [19]: df['is_blend'] = df.varieties.map(set.__len__) >= 2
```

为了后面统计方便，可以再增加两列，用字符串标记该葡萄酒的血统。

```
In [20]: df.loc[df.is_varietal, 'variety_type'] = 'Varietal'
         df.loc[df.is_blend, 'variety_type'] = 'Blend'
         df[['name', 'is_varietal', 'single_variety', 'is_blend', 'variety_type',
             'variety_type']].head()
```

Out[20]:（输出结果如图 4-5-7 所示）

	name	is_varietal	single_variety	is_blend	variety_type	variety_type
0	Riesling Spatlese 2013	True	Riesling	False	Varietal	Varietal
1	Spy Valley - Sauvignon Blanc 2016	True	Sauvignon Blanc	False	Varietal	Varietal
2	Cava Χρυσοχόου 2008	False	NaN	True	Blend	Blend
3	Κτήμα Γεροβασιλείου - Λευκός 2016	False	NaN	True	Blend	Blend
4	Landskroon - Pinotage 2013	True	Pinotage	False	Varietal	Varietal

图 4-5-7　输出结果

数据准备工作完成，下面就要运用各种统计方法研究葡萄酒了。

```
In [21]: ax = df.is_varietal.replace({True: 'Varietal', False: 'Blend'}).value_
             counts().plot('barh')
         ax.set(title='How many wines are varietals and how many blends ?')
```

Out[21]:（输出结果如图 4-5-8 所示）

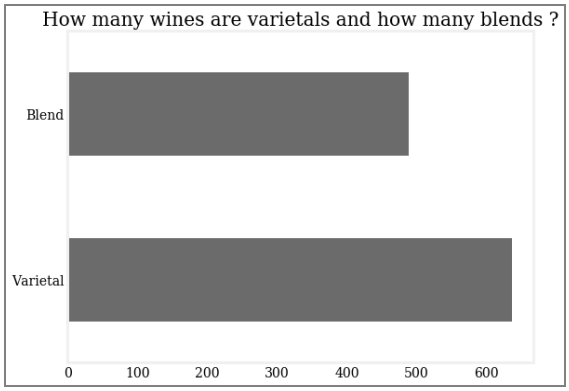


图 4-5-8　一种和多种葡萄酿制的葡萄酒统计

可以明显看出，被标记为“Varietal”的葡萄酒是多数的——看来葡萄酒中“纯种”的也受欢迎。

再深入挖掘一下，是不是能看到宝？

```
In [22]: fig, (ax1, ax2, ax3) = plt.subplots(nrows=3, figsize=(12,14))

         varieties_hist = df[df.is_varietal].varieties.map(lambda x: next(iter(x))).
                          value_counts()
         varieties_hist.head(10).sort_values(ascending=True).plot('barh', ax=ax1)
         ax1.set(title='Most Frequent Varietal Wines (Top 10)');

         varietals_mean_price = df[['single_variety', 'price']].groupby('single_variety'
                                                        ).mean().dropna()['price']
         varietals_mean_price.sort_values(ascending=False
```

```
                                    ).head(10).sort_values(ascending=True
                                                           ).plot('barh', ax=ax2)
        ax2.set(title='Most Expensive Varietals', xlabel='', ylabel='');

        varietals_mean_rating = df[['single_variety', 'avg_rating']].groupby
                                  ('single_ variety'
                                                         ).mean().dropna()['avg_rating']
        varietals_mean_rating.sort_values(ascending=False
                                    ).head(10).sort_values(ascending=True
                                                           ).plot('barh', ax=ax3);
        ax3.set(title='Top Rated Varietals', ylabel='')
```

Out[22]:（输出结果如图 4-5-9 所示）

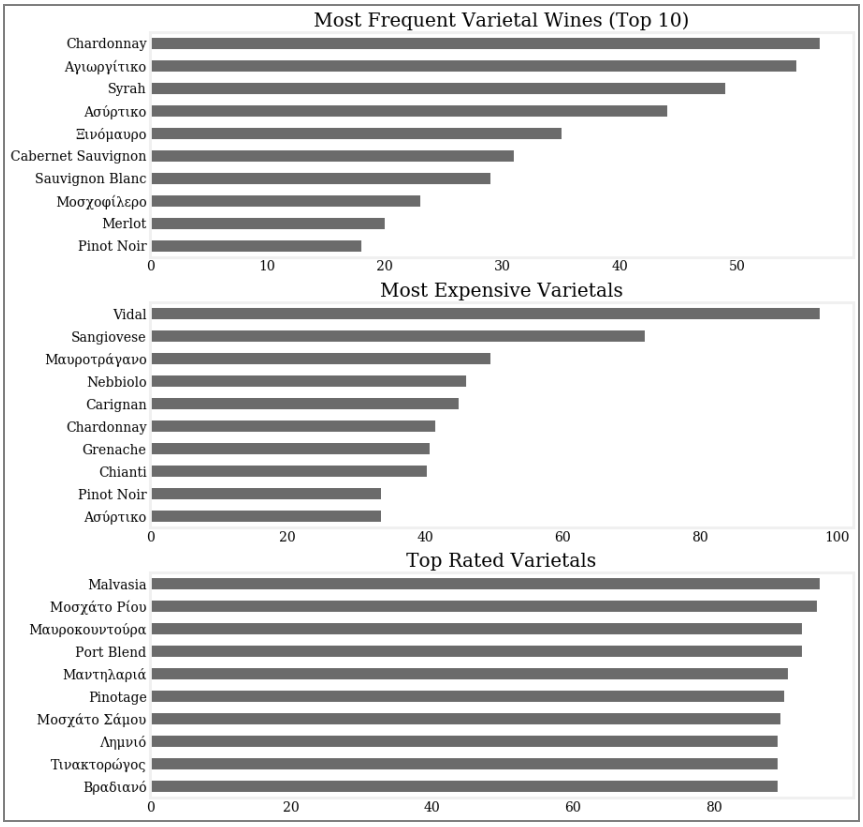


图 4-5-9　单一品类葡萄酒之最

In[22]的代码行数虽然不少，但都是读者可以理解的，并且前面多已解释。不过，笔者还是建议读者逐行阅读，并逐行进行注释。

从图 4-5-9 所示的统计图中能够看出什么？所谓数据分析中的“分析”，就是描述统计结果的含义。

- 使用最多的葡萄品种并不与其所酿酒的价格相符。
- 评级高的酒所用的葡萄品种也不是被众多葡萄酒所采用的。
- 最贵的酒所用的葡萄品种和评级最高的酒所采用的葡萄品种也不重叠。

慢慢琢磨，这个数据挺有意思的——原料、价格、评价，三者居然互不相关。

或许读者还有其他方面的解读。

还是把注意力转移到“混合型”的葡萄酒上。

对于 is_blend 值为 True 的 varieties 中的值，都是由多个词汇组成的集合，见 Out[23]的输出结果。

```
In [23]: df.loc[df['is_blend']==True, ['name', 'is_blend', 'varieties']].head()
Out[23]:（输出结果如图 4-5-10 所示）
```

	name	is_blend	varieties
2	Cava Χρυσοχόου 2008	True	{Merlot, Ξινόμαυρο}
3	Κτήμα Γεροβασιλείου - Λευκός 2016	True	{Ασύρτικο, Μαλαγουζιά}
7	Ψίθυρος Ερυθρός 2010	True	{Cabernet Sauvignon, Λιάτικο, Merlot}
11	Côte Rôtie 2012	True	{Viognier, Syrah}
13	Château d' Yquem 2002	True	{Sauvignon Blanc, Semillon}

图 4-5-10　输出结果

如果要分析 varieties 的值的特点，必须把它看成一个文本，因为它是由多个词汇组成的。于是要进行适当处理，将原来的集合转换为文本。

```
In [24]: docs = df.loc[df.is_blend, 'varieties'].map(lambda x: {s.replace(' ', '_') for
                                                                s in x}
                                                    ).map(lambda x: ' '.join(x))
         docs.head()
Out[24]: 2                          Merlot Ξινόμαυρο
         3                        Ασύρτικο Μαλαγουζιά
         7          Cabernet_Sauvignon Λιάτικο Merlot
         11                           Viognier Syrah
         13                 Sauvignon_Blanc Semillon
         Name: varieties, dtype: object
```

对照 Out[23]和 Out[24]两个输出结果，即可理解 In[24]的写法了。

接下来我们使用 scikit-learn——Python 提供的一个机器学习库——中的工具进行文本特征提取。本例中所提取的是词频特征，因为 varieties 中的词语是标签，没有各种虚词，统计词频已经足够——在很多关于文本特征提取的操作中，TD-IDF 特征是必不可少的，但这里不需要。

注意，如果没有安装 sklearn，则需要先行安装——安装方法你懂的。

```
In [25]: from sklearn.feature_extraction.text import CountVectorizer
         count_model = CountVectorizer(lowercase=False, min_df=.1)
         X = count_model.fit_transform(docs)
         X
Out[25]: <429x8 sparse matrix of type '<class 'numpy.int64'>'
             with 638 stored elements in Compressed Sparse Row format>
```

In[25]的程序即对文本 docs 中的词语进行分析，依次对每个出现在 docs 中的词汇在每一条记录中出现的次数进行统计。从 Out[25]结果可知，在本数据集中，表示葡萄品种的词汇一共有 8 个，即有 8 种葡萄——429×8。

既然是“混合型”，就要看看是某种葡萄和另外哪种葡萄混合的。Florents 为此写了一个函数，最终得到 Out[26]的输出结果。

```
In [26]: from sklearn.feature_extraction.text import CountVectorizer

         def create_coocurrence_df(docs):
            # Source: https://stackoverflow.com/a/37822989
            count_model = CountVectorizer(lowercase=False, min_df=.1) # default unigram
            model
            X = count_model.fit_transform(docs)
            Xc = (X.T * X) # this is co-occurrence matrix in sparse csr format
            Xc.setdiag(0) # sometimes you want to fill same word cooccurence to 0
            ret = pd.DataFrame(Xc.todense())
            ret.index = ret.columns =
                        list(map(lambda f: f.replace('_', ' '), count_model.get_
                             feature_names()))
            return ret

         coocurrence = create_coocurrence_df(docs)
         coocurrence
```

Out[26]:（输出结果如图 4-5-11 所示）

	Cabernet Franc	Cabernet Sauvignon	Chardonnay	Merlot	Pinot Noir	Syrah	Αγιωργίτικο	Ασύρτικο
Cabernet Franc	0	42	0	52	0	2	1	0
Cabernet Sauvignon	42	0	0	85	1	24	21	0
Chardonnay	0	0	0	0	46	0	0	8
Merlot	52	85	0	0	1	29	7	0
Pinot Noir	0	1	46	1	0	1	0	0
Syrah	2	24	0	29	1	0	11	0
Αγιωργίτικο	1	21	0	7	0	11	0	0
Ασύρτικο	0	0	8	0	0	0	0	0

图 4-5-11　输出结果

这里使用了 scikit-learn，本书没有也不会完整介绍 Python 中这个著名的机器学习库——笔者会留在专门的关于机器学习的书中详述。不过，读者可以访问其官方网站（http://scikit-learn.org/）进行初步了解。如果读者遵循了本书从开始到现在一直倡导的研习方法，应该有能力自己研究这个库了。

可以计算一下，哪种葡萄被广泛使用，如图 4-5-12 所示。

```
In [27]: ax = coocurrence.sum().sort_values(ascending=False
                                            ).head(10).sort_values(ascending=True).p
lot(kind='barh')
         ax.set(title='Most Frequent Varieties Appearing in Blends (Top 10)')
Out[27]: [<matplotlib.text.Text at 0x7fbd46afaa58>]
```

笔者特意搜索了“top wine varieties”，居然真的有结果（网址：http://winefolly.com/update/top-wine-varieties/），但不知是否权威，仅供参考。不过跟 Out[27]的结果倒是有部分接近，如图 4-5-13 所示。

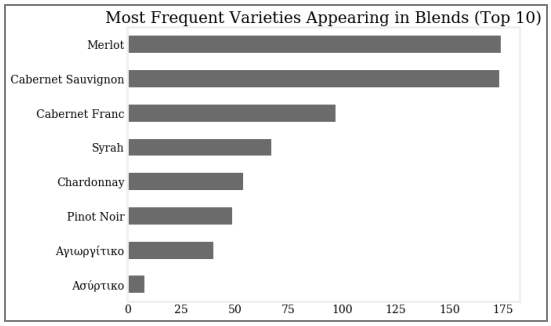


图 4-5-12　统计哪种葡萄被广泛使用

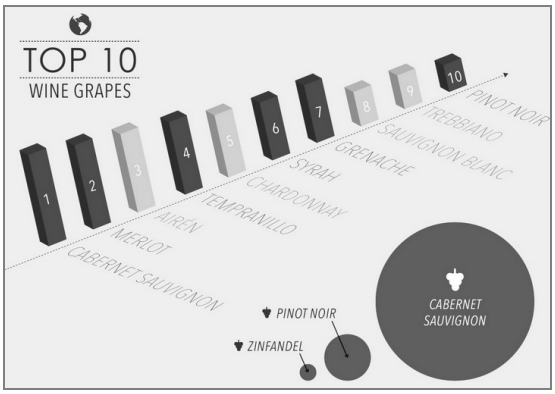


图 4-5-13　笔者搜索的结果

Florents 提供的统计并没有就此结束，他还用热图的方式分别展示了不同品种葡萄的“搭配”情况，更直观地反映了 Out[26]的数据结果。

```
In [28]: ax = sns.heatmap(coocurrence, square=True, annot=True, fmt="d", cmap='Blues')
         ax.set(title='Which varieties are usually blended together ?\n'.title());
         plt.gcf().set_size_inches((8,6))
```

Out[28]:（输出结果如图 4-5-14 所示）

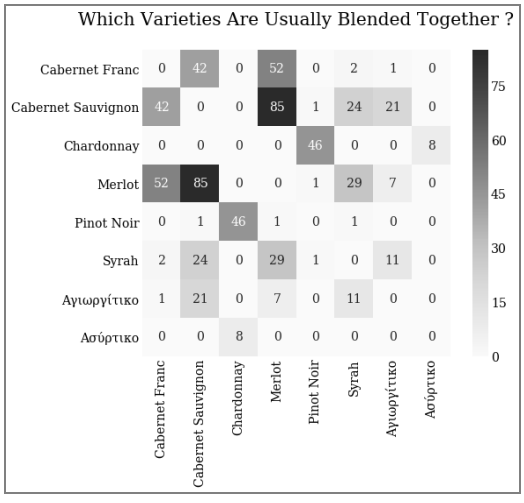

Which Varieties Are Usually Blended Together ?
Cabernet Franc 0 42 0 52 0 2 1 0
Cabernet Sauvignon 42 0 0 85 1 24 21 0
Chardonnay 0 0 0 0 46 0 0 8
Merlot 52 85 0 0 1 29 7 0
Pinot Noir 0 1 46 1 0 1 0 0
Syrah 2 24 0 29 1 0 11 0
Αγιωργίτικο 1 21 0 7 0 11 0 0
Ασύρτικο 0 0 8 0 0 0 0 0
Cabernet Franc
Cabernet Sauvignon
Chardonnay
Merlot
Pinot Noir
Syrah
Αγιωργίτικο
Ασύρτικο
75
60
45
30
15
0

图 4-5-14　葡萄的热图

再细分一下，绘制不同颜色葡萄酒的热图。

```
In [29]: fig, axes = plt.subplots(nrows=3, figsize=(12, 18))
         for c, ax in zip(['Red', 'White', 'Rosé'], axes):
             docs = df.loc[df.is_blend & (df.color==c), 'varieties'].map(
                                       lambda x: {s.replace(' ', '_') for s in x}).
                                       map(lambda x: ' '.join(x))
             cooc = create_coocurrence_df(docs)
             cmaps = {'Red': 'Reds', 'White': 'Blues', 'Rosé': sns.cubehelix_palette(8)}
             sns.heatmap(cooc, square=True, annot=True, fmt="d", cmap=cmaps.get(c),
                         ax=ax)
             ax.set(title='Usually blended varieties for {}\n'.format(c).title());
         plt.tight_layout()
```

Out[29]:（输出结果如图 4-5-15～图 4-5-17 所示）

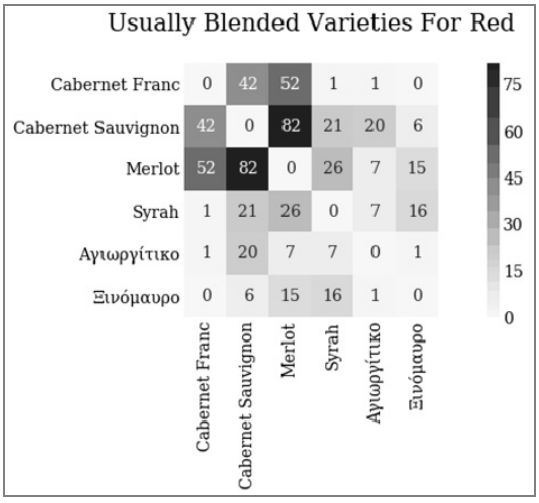


图 4-5-15　不同颜色葡萄酒的热图 1

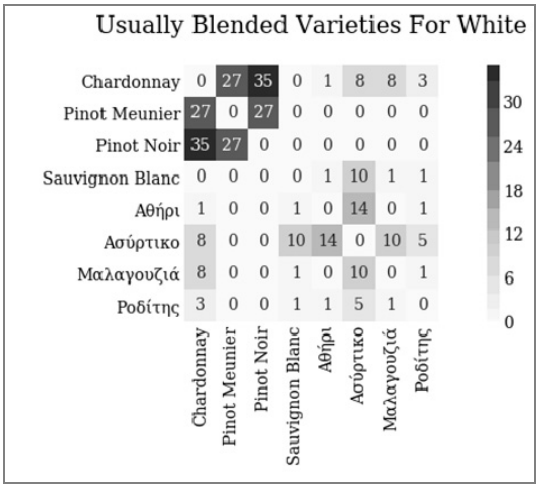

Usually Blended Varieties For White
Chardonnay 0 27 35 0 1 8 8 3
Pinot Meunier 27 0 27 0 0 0 0 0
Pinot Noir 35 27 0 0 0 0 0 0
Sauvignon Blanc 0 0 0 0 1 10 1 1
Αθήρι 1 0 0 1 0 14 0 1
Ασύρτικο 8 0 0 10 14 0 10 5
Μαλαγουζιά 8 0 0 1 0 10 0 1
Ροδίτης 3 0 0 1 1 5 1 0
Chardonnay
Pinot Meunier
Pinot Noir
Sauvignon Blanc
Αθήρι
Ασύρτικο
Μαλαγουζιά
Ροδίτης
30
24
18
12
6
0

图 4-5-16　不同颜色葡萄酒的热图 2

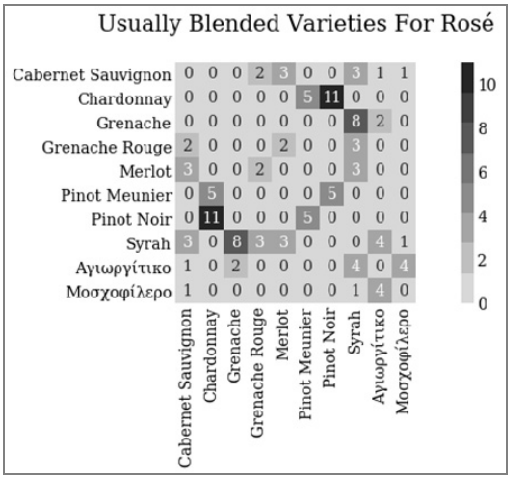

Usually Blended Varieties For Rosé
Cabernet Sauvignon 0 0 0 2 3 0 0 3 1 1
Chardonnay 0 0 0 0 0 5 11 0 0 0
Grenache 0 0 0 0 0 0 0 8 2 0
Grenache Rouge 2 0 0 0 2 0 0 3 0 0
Merlot 3 0 0 2 0 0 0 3 0 0
Pinot Meunier 0 5 0 0 0 0 5 0 0 0
Pinot Noir 0 11 0 0 0 5 0 0 0 0
Syrah 3 0 8 3 3 0 0 0 4 1
Αγιωργίτικο 1 0 2 0 0 0 0 4 0 4
Μοσχοφίλερο 1 0 0 0 0 0 0 1 4 0
Cabernet Sauvignon
Chardonnay
Grenache
Grenache Rouge
Merlot
Pinot Meunier
Pinot Noir
Syrah
Αγιωργίτικο
Μοσχοφίλερο
10
8
6
4
2
0

图 4-5-17　不同颜色葡萄酒的热图 3

至此，葡萄酒的数据分析完毕。

敲击着反应都有点迟钝的键盘，引入 Python 各种计算模块，“计算着梦想和现实之间的差距”——“在钢筋水泥的丛林里，在呼来唤去的生涯里”。

4.6　应用本福特定律

在解释本福特定律之前，先看一个统计实例。

根据《维基百科》的“国家人口列表”词条的数据（2017 年 12 月 4 日的数据，读者阅读的时候，或许该网站会有所更新），建立一个 CSV 文件。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt

        pop_world = pd.read_csv(
                  "/home/qiwsir/Documents/DataAnalysis/chapter04/population_world.csv")
        num = pop_world['number'].map(str).map(lambda x: x[0])    #①
        num = num.map(int)
        counts = num.value_counts() / 228     #②
        r = pd.DataFrame({"number": counts.index, 'percent': counts.values})
        r.sort_values('number')
Out[1]:     number    percent
        0       1    0.311404
        2       2    0.131579
        1       3    0.131579
        4       4    0.100877
        3       5    0.105263
        5       6    0.070175
        8       7    0.048246
        6       8    0.052632
        7       9    0.048246
```

```
In [2]: plt.bar(r['number'], r['percent'])
        plt.grid(True)
        plt.xticks(r['number'])
```

Out[2]:（输出结果如图 4-6-1 所示）

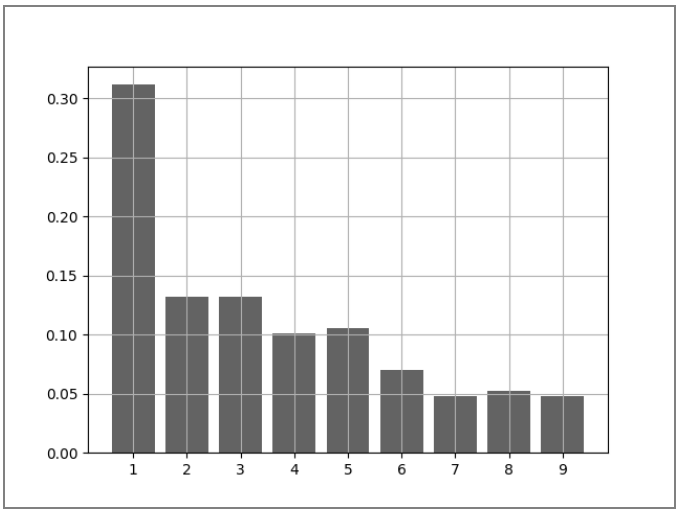


图 4-6-1　In[2]的输出结果

In[1]中的语句①实现将人口数字中的第一个有效数字取出来，并在语句②中统计各个数字的出现频率，如 Out[1]中的 percent 列所示。然后用 In[2]的程序画图。

最终我们发现，数字 1 出现的比例最高，并且大约是总数的三分之一。

这种现象在其他的某些情况中也存在。

再比如，有一个名叫西蒙·纽康伯的加拿大天文学家在 1881 年发现对数表中以 1 起首的数那几页较其他页更加破烂——这可能是虚构的。后来，1938 年，美国电器工程师兼物理学家法兰克·本福特也发现了这个现象，并且他又收集了更多各种完全不相同的数据进行分析检验，于是跟你预料的一样，他发现了一个以自己名字命名的定律“本福特定律”。

从这个故事中得到的教育意义，读者自己总结吧。当然，总结不出来是最正常的了。

引用《维基百科》对本福特定律的表述如下：

在 b 进位制中，以数 n 起头的数（第一个有效数字）出现的概率为 $\log_b(n+1)-\log_b(n)$。

表 4-6-1 列出了十进制首位数字的出现概率（小数点后保留一位）。

表 4-6-1　本福特定律数字概率表

d	1	2	3	4	5	6	7	8	9
p	30.1%	17.6%	12.5%	9.7%	7.9%	6.7%	5.8%	5.1%	4.6%

将表 4-6-1 和 Out[1]的输出结果对照，发现它们比较接近，可以说各个国家和地区的人口数字的第一个有效数字符合本福特定律。

```
In [3]: benford = pd.DataFrame({ "number": [1,2,3,4,5,6,7,8,9],
                                 'percent': [0.301, 0.176, 0.125, 0.097, 0.079,
                                             0.067, 0.058, 0.051, 0.046]})
```

```
        position = benford['number'] - 0.2
        plt.bar(position, r['percent'], width=0.4, label='population', color='b')
        plt.bar(position+0.4, benford['percent'], width=0.4, label='benford', color='y')
        plt.xticks(benford['number'])
        plt.grid(True)
```

Out[3]:（输出结果如图 4-6-2 所示）

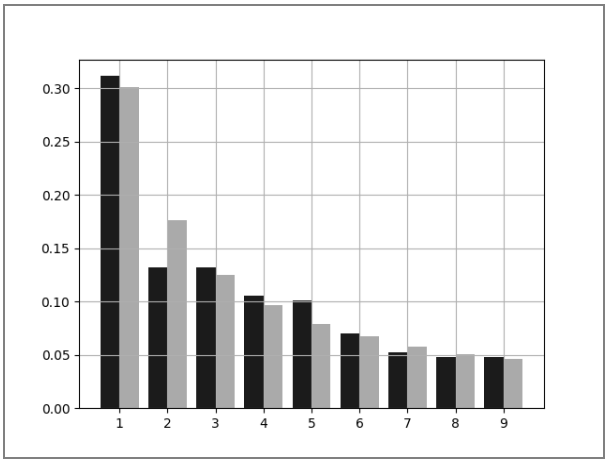


图 4-6-2　人口数字的分布

从 Out[3]的输出结果中更明显地看出，人口数字的分布符合本福特定律。不仅如此，符合本福特定律的其他领域的数据还有很多，比如海拔高度、股票价格、会计数据等。曾有报道，2001 年破产的美国最大的能源交易商安然公司，就有高层管理人员涉嫌做假账的传闻，该公司 2001 年到 2002 年所公布的每股盈利数字不符合“本福特定律”，很有可能是经过人工修改的。

除用类似 Out[3]那样的结果，通过观察图示说明某数据集符合本福特定律外，我们还能够应用统计学的知识，计算相关系数。

下面研究苹果公司的股票。

可以继续使用 4.1 节 In[1]的方法得到苹果公司的股票数据，也可以到相关网站下载，比如 Yahoo 财经就可以满足此需求（https://finance.yahoo.com/quote/AAPL）。下面演示用的数据就是从这里下载的 CSV 文件，如果读者在前面没有通过 Pandas 爬取到代码，就可以使用这种方法下载。前面没有告知这种做法，是因为“好菜”总是后上的。

```
In [4]: aapl = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter04/AAPL.csv")
        aapl.count()
Out[4]: Date          485
        Open          485
        High          485
        Low           485
        Close         485
        Adj Close     485
        Volume        485
        dtype: int64
```

In[4]读入的数据集中一共有 485 条记录。下面验证其中的 aapl['Volume']数据是否符合本福特定律。

```
In [5]: def calc_firstdigit(dataset):
        fdigit = [str(value)[:1] for value in dataset]
        distr = [fdigit.count(str(i))/float(len(dataset))*100 for i in range(1, 10)]
        return distr
In [6]: import math
        benford = [math.log10(1+1/float(i))*100.0 for i in range(1,10)]
```

In[5]是专门获得第一个有效数字的函数，返回一个列表；In[6]也是应用最基本的 Python 知识得到本福特数据的。之所以写这两个程序，是为了跟前面稍微区别一下，有人说写程序如同写诗，在诗句中，最好别出现太多的重复词语。

```
In [7]: aapl_volume = aapl['Volume']
        aapl_calc = calc_firstdigit(aapl_volume)     #调用 In[5]的函数

        fig, ax = plt.subplots()
        ax.plot([1,2,3,4,5,6,7,8,9], aapl_calc, label='AAPL')
        ax.plot([1,2,3,4,5,6,7,8,9], benford, label='Benford')
        plt.legend()
        plt.grid(True)
Out[7]:（输出结果如图 4-6-3 所示）
```

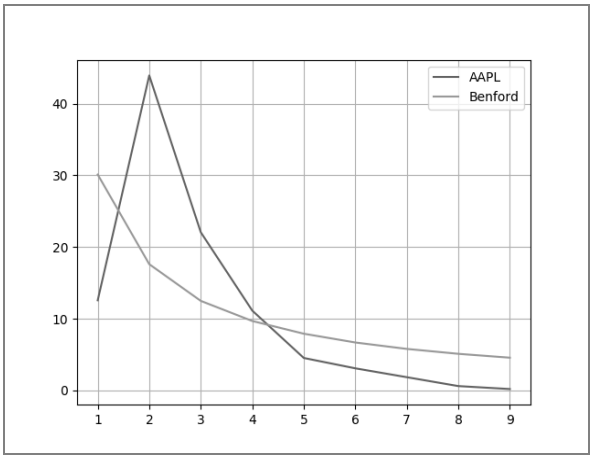


图 4-6-3　AAPL 股票数据的第一个有效数字分布

从 Out[7]的结果看，貌似不如 Out[3]结果吻合得好。

下面再计算一下两者的皮尔森相关系数。

```
In [8]: from scipy.stats import pearsonr
        r_row, p_value = pearsonr(benford, aapl_calc)
        r_row, p_value
Out[8]: (0.52842273214202418, 0.14359938153082782)
```

果然，其相关系数为 0.53，有点勉强。是不是说明我们所使用的数据集不符合本福特定律呢？

还不能过早下结论。再看一个数据集，还是苹果公司的股票，只不过这次是从 1983 年开始的。

```
In [9]: aapl_long = pd.read_csv("/home/qiwsir/Documents/data_analysis/chapter04/AAPL-
                                long.csv")
        aapl_long_volume = aapl_long['Volume']
        aapl_long_volume.count()
```

```
Out[9]: 8807
```

数据量比原来多了不少，结果呢？

```
In [10]: aapl_long_calc = calc_firstdigit(aapl_long_volume)
         r2_row, p2_value = pearsonr(benford, aapl_long_calc)
         r2_row, p2_value
Out[10]: (0.98334614975620205, 1.9310143433410221e-06)
```

In[9]得到的数据与本福特的相关系数达到了 0.98，说明 aapl_long_volume 所引用的数据非常符合本福特定律。

再画图看看效果。

```
In [11]: fig, ax = plt.subplots()
         ax.plot([1,2,3,4,5,6,7,8,9], aapl_long_calc, label='AAPL')
         ax.plot([1,2,3,4,5,6,7,8,9], benford, label='Benford')
         plt.legend()
         plt.grid(True)
Out[11]:（输出结果如图 4-6-4 所示）
```

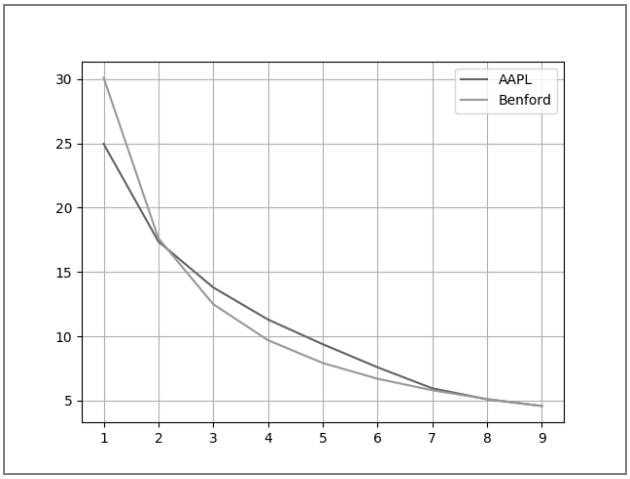


图 4-6-4　AAPL 股票数据的分布与本福特定律

本福特定律胜利了。

此规律的应用还在继续。

- 1972 年，Hal Varian 提出用此定律检查某些公共计划的经济数据是否有欺瞒之处。
- 1992 年，Mark J. Nigrini 在其博士论文 *The Detection of Income Tax Evasion Through an Analysis of Digital Frequencies*（Ph.D. thesis. Cincinnati, OH: University of Cincinnati, 1992.）中提出用它检查是否有伪账。
- 2009 年，西班牙数学家发现素数数列中每个素数的首位数字的分布可以被用来描述本福特定律（以上三点根据《维基百科》“本福特定律”词条整理）。
- 物理学家发现，统计物理中的 Boltzmann-Gibbs 分布、Bose-Einstein 分布、Fermi-Dirac 分布，也基本上满足本福特定律（根据 https://www.guokr.com/article/520/整理）。

如果读书学习也符合本福特定律多好呀，专拣着数字“1”开头的页码看，就能掌握整本书

知识的三分之一——最好只选最多 200 页的书读，更划算。

在做什么梦？

4.7 制作词云

“词云”，又称为“标签云”“文字云”等，就是将文本中指定条件的词汇以图示的方式展示出来，例如图 4-7-1。

Afghanistan Algeria Angola Argentina Australia Bangladesh Belarus Belgium Benin Bolivia
Brazil Bulgaria Burkina Faso Burma Burundi Cambodia Cameroon Canada Chile
China Colombia Congo (Kinshasa)
Dominican Republic Ecuador Egypt El Salvador Eritrea Ethiopia Finland France Germany Ghana Greece Guatemala
India Indonesia Iran Iraq Italy
Guinea Haiti Honduras Hong Kong S.A.R. Hungary
Japan Jordan Kazakhstan Kenya Korea, North Korea, South Kyrgyzstan Laos Lebanon Libya Madagascar Malawi Malaysia Mali
Mexico Morocco Mozambique Nepal Netherlands Nigeria
Pakistan Peru Philippines Poland Portugal Romania Russia Rwanda
Saudi Arabia Senegal Serbia Somalia South Africa Spain Sri Lanka Sudan Sweden Switzerland Syria Taiwan Tajikistan
Tanzania Thailand Tunisia Turkey Uganda Ukraine United Kingdom
United States Uzbekistan Venezuela Vietnam Yemen Zambia Zimbabwe
Scale:
1,400 mil
1,000 mil
600 mil
400 mil
200 mil
0 mil
Data file: Population by Country 2007
Data source: US Census Bureau International Data Base
This data set has not yet been rated.

图 4-7-1　各国人口数量词云图（来自《维基百科》）

词云也是实现数据可视化的一种途径，例如从图 4-7-1 中可以非常直观地看出各国人口数量的相对多少。

这种图，我们也能做。

先安装第三方库 wordcloud（网址：https://github.com/amueller/word_cloud）。

```
$ sudo pip3 install wordcloud
```

安装完毕，绘制一个简单的图。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt

        from wordcloud import WordCloud, STOPWORDS

        file_name ="/home/qiwsir/Documents/ DataAnalysis/chapter04/hertz.txt"
        with open(file_name) as f:
            text = f.read()

        stop_words = set(STOPWORDS)
        word_cloud = WordCloud(background_color="white",
                               max_words=200, stopwords=stop_words)
        word_cloud.generate(text)

        plt.imshow(word_cloud, interpolation='bilinear')
        plt.axis("off")
```

```
Out[1]: (-0.5, 399.5, 199.5, -0.5)
```

（输出结果如图 4-7-2 所示）

图 4-7-2　简单的词云

从得到的词云中可以明显地看出，Hertz、family、University、physics 等单词是出现频率较高的。

请读者关注 In[1]中的 STOPWORDS。在文本中，有些词语出现的频率虽然也较高，但不是绘制词云时所需要的，这种词语就统一放到 STOPWORDS 中。

```
In [2]: stop_words = set(STOPWORDS)
        stop_words.add("laoqi")
        stop_words
Out[2]: {'a',
        …    #省略部分内容
        'laoqi',
        …}
```

除 STOPWORDS 默认的词汇外，还可以根据需要向里面添加新的词汇。

以上是英文词云，中文的要比英文的复杂。

下面介绍一个中文词云示例。

先准备一些中文内容。不再使用类似于 In[1]中的那种方式了，而是要找一些更贴近实际业务的中文数据——微博内容。

当然，因为本书不是讲解爬虫的，所以不说明数据的获取过程，只是展示数据集结果。

下面所用的数据集是笔者在 weibo.com 网站输入“python”进行搜索，然后用爬虫技术搜索到的微博内容。

```
In [3]: weibo_data = pd.read_csv(
                    "/home/qiwsir/Documents/DataAnalysis/chapter04/weibo_data.csv")
        weibo_data.count()
Out[3]: id                978
        weibo_id          978
        weibo_cont        978
        weibo_img         413
        weibo_video       6
        repost_num        978
        comment_num       978
        praise_num        978
        uid               978
```

```
is_origin         978
device            971
weibo_url         978
create_time       978
comment_crawled   978
repost_crawled    978
dtype: int64
```

当然，数据量也不是很大，但聊胜于无。

汉语和英语的区别不小，在自然语言处理（NLP）中不可忽视的一个差别是“一句英语中，单词之间有空格；一句汉语中，字和字之间没有空格”。要处理汉语，就需要把“句子”分成若干个“词”——分词——本书使用 jieba 第三方库（https://github.com/fxsjy/jieba）。

```
$ sudo pip3 install jieba
```

特别鸣谢：以下程序编写和调试得到了好友“当当虾”的鼎力支持。

在写代码之前，先做点准备工作。

创建一个名为 weibo 的子目录，在里面创建名为 stopwords.txt 的文件。在 In[1]中，我们已经使用过 STOPWORDS 这个由 wordcloud 库提供的“停用词”数据集，但是里面都是英文的，现在要处理中文，也要创建一个中文的“停用词”数据集。于是使用一个文本文件，可以随时更改里面的停用词。

此文本文件的基本内容如下：

```
的
了
和
...
更多
详情
网页
链接
一个
使用
什么
可以
分享
看见
```

文件中每个被停用的词占据一行。

接下来在 weibo 目录中创建一个.py 文件（test.py），然后在这个文件中逐步写入一些程序。

```
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt

import os
import re

from wordcloud import WordCloud
import jieba

app_path = os.path.dirname( os.path.dirname(__file__) )

stop_words_txt = os.path.join(app_path, 'stopwords.txt')

with open(stop_words_txt) as f:
    for line in f:
        STOP_WORDS.add(line[:-1])
```

依据已经创建的停用词文本文件创建停用词集合，然后读入所得到的微博数据，我们要处理的词语来自微博的文章内容，就需要对其进行适当的处理。

```
file = './weibo_data.csv'
weibo_datas = pd.read_csv(file)

content_weibo = weibo_datas['weibo_cont']
content_weibo_cut = content_weibo.apply(jieba.cut)    #①
```

语句①的作用就是对每条微博内容使用 jieba.cut()进行分词。注意，分词结果加入列表中，得到了一个生成器对象。

读者可以通过循环语句，把分词结果打印出来，会发现得到的词汇五花八门，有很多是后面分析不需要的，比如“的、地、得”之类，所以要对这些词汇进行整理——数据清洗是每个数据工程师必须要做的事。

为此，先写一个函数。

```
def filter_words(words, stop_words):
    result = [w for w in words if w not in stop_words and len(w)>1 /
                                                          and re.match("^[\u4e00-\u9fa5]
                                                          {0,}$", w)]
    return result
```

这个函数的主体就是那个列表解析式，请关注其中的三个条件，通过它们对微博的分词结果进行筛选。

得到了分词结果，对每个词汇在整个微博数据集中出现的次数进行计数，然后用计数结果来绘制词云——与前述绘制方法有别。

```
word_freq = dict()

for one in content_weibo_cut:
    row = filter_words(list(one), STOP_WORDS)
    if row:
        for word in row:
            word_freq[word] = word_freq.get(word, 0) + 1
```

变量 word_freq 所引用的对象是一个字典，它将以词为 key，以该词的计数为 value，并将

其用于②来生成词云。

```
font_path = os.path.join(app_path, 'simkai.ttf')
```

这是专门针对汉字的字体，必不可少，因为词云默认的字体不显示汉字。

```
word_cloud = WordCloud(font_path=font_path,  #设置字体
                       background_color="white",  #背景颜色
                       max_words=200,  #词云显示的最大词数
                       max_font_size=100,
                       random_state=42,
                       width=600, height=400, margin=2,
                      )
word_cloud.generate_from_frequencies(word_freq)    #②

plt.figure()
plt.imshow(word_cloud)
plt.axis("off")

plt.show()
```

程序编写完毕，保存、运行。是否能得到图 4-7-3 所示的结果？如果不能，请耐心、认真地检查，不放过任何一个拼写。

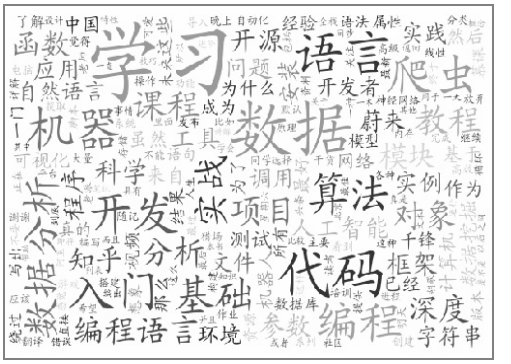


图 4-7-3　中文词云

图 4-7-3 是搜索“Python”之后，分析相应微博内容而得到的词云，从这个结果中你能读出什么信息？谈论“Python”的人也在谈论着：

- 学习——很多人刚刚开始学习这种语言；
- 数据——他们或者在用它处理数据，或者期望能够学会了它之后去处理数据；
- 入门——真的有不少人刚刚开始学习；
- 爬虫——这个东西受到了关注，或许是要学习，或许是用它做什么事情。

这只是笔者的解读罢了，还有很多方面的解读。不过，这不是本书关注的重点。

本书的重点是让读者学会数据处理方法。

虽然本书的目标是带领读者入门数据分析，但在入门的走廊上，也要打开天窗看看远处的风光——这就是第 5 章的目的。

第5章

机器学习

当“机器学习”这个词语几乎已经成为街头巷尾、茶余饭后的谈资时，一本号称“数据分析”的书必须要有这么一章，否则就太不追赶时尚了。但是，本章充其量算是机器学习的入门概览，或者说是通过几个实例，让读者初步领略一下机器学习的魅力，从而引导读者进驻这个领域。

既然如此，那么就不会有关于机器学习原理方面的介绍了。这可能让读者感觉不完善，幸而市面上早就有了各种版本的相关专著，如果读者有意深入，可以找来阅读。并且，也可以说是预告，或者是吊胃口吧，后续笔者也有计划编写一本融合“原理和实战”的机器学习方面的书。

除追赶时尚外，本书在最后一章提供机器学习的有关示例，也是因为机器学习与前面所讲述的各章内容密切相关。前面讲述的数据统计，目的就是要通过数据找出规律。因为只要知道了规律，我们就可以按照规律办事情，进而达到某种目的。

机器学习也是找规律的一种方法，可以看作是前述几章的延续。

从本章开始，是一个数据分析师“武功”提升的契机，掌握了机器学习的工程师，是时代的骄子。

5.1 线性回归

人天生就有一种探索世界的欲望。在探索的过程中，形成了很多方法，其中最常用的就是创建一个简单模型，然后用这个模型来研究事物。

在诸多模型中，线性模型是我们更喜欢的，因为它相对比较简单，并且适用面比较广。比如我们喜欢用“龙生龙、凤生凤、老鼠的儿子会捣洞”这种模型来预测后代的未来。当某个模型在实验或实践上被发现有问题时，就会有天才提出另外的模型；天才毕竟不同于凡人，往往他们提出某种模型，通常很久之后，我们这些凡人才发现那个东西有用。

例如，下面的记载中就是一个天才的故事。

据传，1666 年，在英国北部林肯郡的一个小村子里，有一个百无聊赖的青年，坐在苹果树

下面打盹，秋日午后的阳光洒在身上，暖暖的。突然，树上掉下来了一个苹果，正好砸在了这个青年的头上，他为之一振，灵光乍现，立刻起身，看看地上的苹果，仰头望望天空——太阳、白云和掠过的飞鸟，青年张开双臂，长啸一声——万有引力定律——当然，他说的是英语。

这个青年名叫伊萨克·牛顿。更有诗人对牛顿的不吝赞美：

自然和自然规律隐没在黑暗中；

上帝说，让牛顿去吧！

于是一片光明。

幸亏他不嘴馋，如果他只顾着吃苹果，我们还生活在“黑暗”中呢！

牛顿通过对自然、实验的观察，运用数学工具，发现物理规律（以定律、定理等形式表述），甚至到现在，我们还在享用这些物理规律的红利。

本节所介绍的“线性回归”乃至整个机器学习的基本思路都遵循了牛顿时代所开创的思想方法。

终于回到正题了。

1．线性回归简述

“回归”分析是统计学和机器学习中常用的一种方法，英文是 Regression。据《维基百科》解释，这个词最早是由弗朗西斯·高尔顿（Francis Galton）开始使用的。他发现父母的身高虽然会遗传给子女，但子女的身高却有逐渐“回归到中等（人的平均值）”的现象。

虽然现在的“线性回归”已经不是“回归到中等”了，但依然使用这个词语。

现代统计学和机器学习语境中的“回归”是指通过对数据（自变量和因变量）的分析，找到某种规律，建立数学模型（自变量和因变量之间的函数关系）。

到目前为止，人类中的天才们已经为“回归”提供了几种数学模型，比如“线性模型”，通常就把利用这种模型所进行的分析称为“线性回归”。此外还有 Logistic Regression、Polynomial Regression、Ridge Regression 等——姑且仅给出英文名称。

线性回归，因其简单、适用面广，一直以来被认为是统计学和机器学习入门必学的。所以，本节就以线性回归为例，讲解机器学习的一般研究思路。

发展于伽利略、牛顿年代的一套科学研究方法，经过众多大师的锤炼，日臻完善，并从物理学拓展到其他学科门类——物理是科学的基础，绝不是自吹自擂。这套方法对线性回归乃至于机器学习都适用，简而言之，其步骤如下。

第一步：收集数据。可以通过观测、实验，利用各种测量工具收集被研究对象的相关数据，形成“数据集”。当然，在网络环境中，还可以通过“网络爬虫”、API 等方式获得数据。

第二步：归纳规律。归纳规律的方法有很多种，一般需要与专业知识相配合。比如，在坐标系中画出数据集的散点图，然后找规律——这个规律通常用数学表达式来体现。“一种科学，只有成功地运用数学时，才算达到了真正完善的地步”（据说是马克思说的，但没有查找到出处），能够用数学表达的规律，不仅严谨，也体现着简洁美——这种美需要有专业知识才能体验。

第三步：用新的数据验证规律，实际应用也是验证。

第四步：修正或者突破原有规律，发现新的规律。

以上流程循环往复，不断推动着科学的发展——推荐读者阅读《量子力学史话》这本书，它会让读者更深刻地体验到上述循环过程的特点和波澜壮阔、惊心动魄的科学发展过程。

在机器学习中，不论使用哪种算法，基本上都是按照上述流程进行的。接下来读者会亲历这个过程。

2．研究一个古老定律

还记得在 3.3 节中所画的散点图吗？我们根据散点图画出了一条直线，用这条直线代表散点图中各个数据点的分布规律。如果用数学方式来表示图中的直线，可以写成如 $y = a + bx$，具体到每个数据点，相对于直线方程还可以有一定的误差。

函数 $y = a + bx$ 就是线性模型的一种函数——这种回归叫作“线性回归”（英文是 Linear Regression），通常其因变量（y）是连续的，自变量（x）可以是连续的，也可以是离散的。

有数学表达式之后，就可以用它来检验新的自变量数据，预见因变量的变化。但是其中的 a、b 是需要通过已知数据集计算得到的。确定系数 a、b 的过程称为“拟合”，“最小二乘法”是一种常用的拟合方法——关于“最小二乘法”的数学问题，此处默认读者已经了解一二。

最后要注意，拟合出来的不一定都是直线，也不一定都是一次函数的形式。通常可以这样来表示拟合所得函数：

$$y = \beta_0 + \beta_1^{\mathrm{T}} X$$

通过已知的 X 和 y 数据，求得 β_0 和 β_1，此线性模型就得以确定。这个过程，用机器学习的行话说，就是“学得 β_0 和 β_1”。

下面通过一个示例，具体演示如何学得 β_0 和 β_1，也就是使用线性回归模型对一些已知数据进行研究，最终得到线性模型表达式。

笔者已经准备好了一些数据，放在一个 CSV 文档中（读者可以在本书的代码仓库中下载），暂时不揭密这些数据是怎么来的，先看看大概的模样——假设“收集数据”工作已经完成。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        filename = "/home/qiwsir/Documents/DataAnalysis/chapter05/snell.csv" #数据地址
        datas = pd.read_csv(filename)
        datas.head()
Out[1]:       alpha       belta
        0    0.001745     0.001164
        1    0.017577     0.011717
        2    0.033408     0.022270
        3    0.049239     0.032819
        4    0.065071     0.043364
```

现在开始找 alpha 和 belta 两列数据所符合的线性模型。

在机器学习中，为了应对“总结规律”和“验证规律”两个步骤，通常要把已经获得的数据集划分为两部分，并相应地冠以“训练集”和“测试集”的名称，用“训练集”来归纳出一

个规律，然后用“测试集”检验这个规律。当然，更全面的检验还有待于此规律的实际应用。

对于“训练集”和“测试集”的划分，不能简单地“五五开”（为什么？请读者思考）。

```
In [2]: from sklearn.model_selection import train_test_split
        X_train, X_test, Y_train, Y_test = train_test_split(
                                           datas['alpha'], datas['belta'],
                                           test_size=0.4, random_state=40)
```

train_test_split()是 sklearn.model_selection 模块中提供的随机划分样本数据的函数，其中参数 test_size 表示训练集数据量占整个样本数据量的比例，random_state 是随机数种子。

```
In [3]: datas['alpha'].count(), X_train.count(), X_test.count()
Out[3]: (100, 60, 40)
```

从 Out[3]中可以看到，In[2]的操作的确将原来的数据划分为两部分，训练集占 60%。后面就要从训练集的数据中归纳出规律——一种线性模型。

先画散点图，看看这些数据的基本分布情况。

```
In [4]: plt.scatter(X_train, Y_train)
Out[4]:（输出结果如图 5-1-1 所示）
```

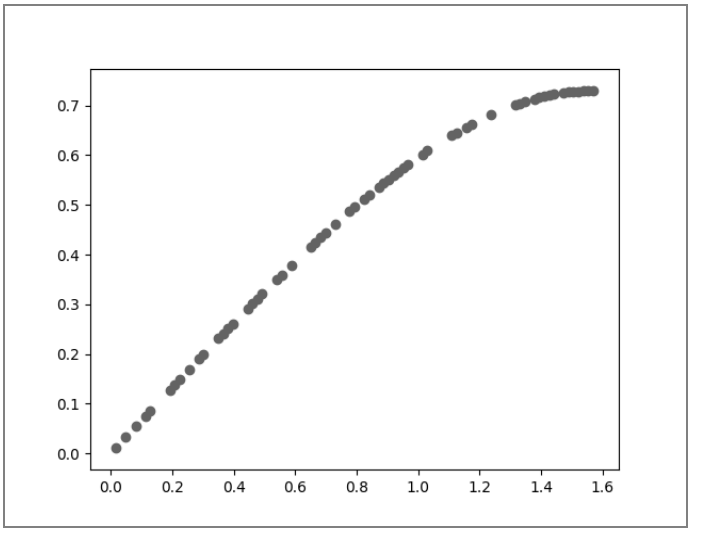


图 5-1-1　实验数据的散点图

观察图 5-1-1 所示的散点图，先运用初中代数知识猜测一下，图中能够拟合出来的曲线可能是什么函数？

这就是建立假设——大胆假设。

姑且假设是 $y = b + ax$ 模型。或许读者认为笔者选择的模型明显有问题，应该是二次曲线。在此，暂且搁置争论，请继续阅读。

按照前面所述，我们要做的就是确定 b 和 a 的值。

作为一名物理系的学生，笔者会按照下面这样的步骤做：

① 在图 5-1-1 中画一条直线，让图中的数据点比较均匀地分布在这条线的两侧；

② 在直线上确定两个点，通过其横、纵坐标，根据斜率的定义计算得到 a 的值；

③ 延长直线，找到其与纵坐标的交点，从而确定 b 的值。

这是物理老师教的。

其中第①步，就是“最小二乘法”，即各数据点到直线距离之和最小。

在机器学习中，我们依然使用“最小二乘法”来拟合曲线，与上述物理系学生做法不同的是，要使用一些专门的模块或第三方库——这是 Python 的重要特点，“轮子”众多，就“最小二乘法”而言，Python 中不止提供了一个“轮子”。不过“饭总要一口一口吃”，还是先从简单、常用的开始吧——Statsmodels。

Statsmodels 是一个 Python 库，它提供多种统计计算方法和模型，完整的文档内容可以阅读文档（https://pypi.python.org/pypi/statsmodels）。

当然，在使用前要安装，这是惯例。

```
$ sudo pip3 install statsmodels    #sudo pip install statsmodels
```

安装之后，按照下面的方法来使用它。

```
In [5]: import statsmodels.api as sm    #①
        X = pd.DataFrame({"c": np.ones(60).T, "x": X_train})    #②
        model = sm.OLS(Y_train, X)    #③
        result = model.fit()    #④
        result.params    #⑤
Out[5]: c    0.075643
        x    0.471903
        dtype: float64
```

In[5]代码虽然不多，但是展示了创建模型的常用方法。

①引入 statsmodels.api，并更名为 sm，这也是惯例——要尽可能地符合惯例。

我们已经假设此模型符合函数 $y = b + ax$，②中创建了包含常数项和训练集 X_train 的数据，np.ones(60)表示常数项，如果 b 等于 0，则无常数项。

③中调用了 OLS 类，它的全称是“普通最小二乘法”（Ordinary Least Squares，OLS），传入训练集 Y_train 和 X，表示要创建一个以 OLS 拟合数据的线性回归模型示例——“万物皆对象”，这个模型也是对象。

④中的 fit()是模型的一个方法，它的作用效果是对③所创建的对象进行“训练”，得到一个具体的拟合结果——还是对象，用 result 变量引用，可以简称为“result”对象。

特别提醒读者，如果从 fit()这个方法的英文含义理解“训练”，应该是能够反映其本质的。fit 的基本含义是“to be the right shape and size for sb/sth”，这里可以理解为要让所建立的模型对象相对于所传入的数据而言“to be right”。

result 对象有一些属性和方法，比如⑤，通过 result.params 属性得到了各项的系数，即 Out[5]输出结果。由此，我们确定表示这条直线的函数式为 $y = 0.076 + 0.472x$。

这个模型怎么样？最直观的方式是把它的图线画出来，与散点图比较。

```
In [6]: coe = result.params
        fig, ax = plt.subplots()
        ax.scatter(X_train, Y_train)
        x = np.linspace(0, 1.8, 100)
        y = coe['c'] + coe['x']*x
        ax.plot(x, y, color='red')
Out[6]:（输出结果如图 5-1-2 所示）
```

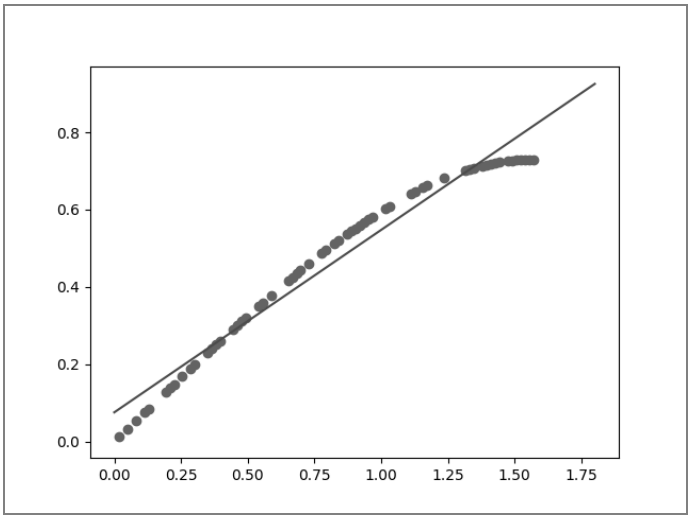


图 5-1-2　拟合为一次函数

从总体上看，似乎做到了“数据点比较均匀地分布在直线两侧”。但很明显，这个结果还有改进的余地。笔者前面说过要搁置争议，故意演示一下拟合直线的过程，目的在于让读者通过简单的模型，理解拟合的基本过程。

下面就假设曲线是二次函数，即 $y = b + a_0 x + a_1 x^2$ 模型——如你所愿吗？基本操作步骤与前面雷同。

```
In [7]: XX_train = pd.DataFrame({"c": np.ones(60).T, "x":X_train, "x**2":
                                 X_train*X_train})
        result2 = sm.OLS(Y_train, XX_train).fit()
        result2.params
Out[7]: c      -0.022771
        x       0.820724
        x**2   -0.210711
        dtype: float64
```

由 Out[7]的结果可知，二次函数模型应该是 $y = -0.022 + 0.821x - 0.211x^2$。这个模型怎么样？

```
In [8]: coe = result2.params
        fig, ax = plt.subplots()
        ax.scatter(X_train, Y_train)
        x = np.linspace(0, 1.8, 100)
        y = coe['c'] + coe['x']*x + coe['x**2']*x*x
        ax.plot(x, y, color='red')
Out[8]:（输出结果如图 5-1-3 所示）
```

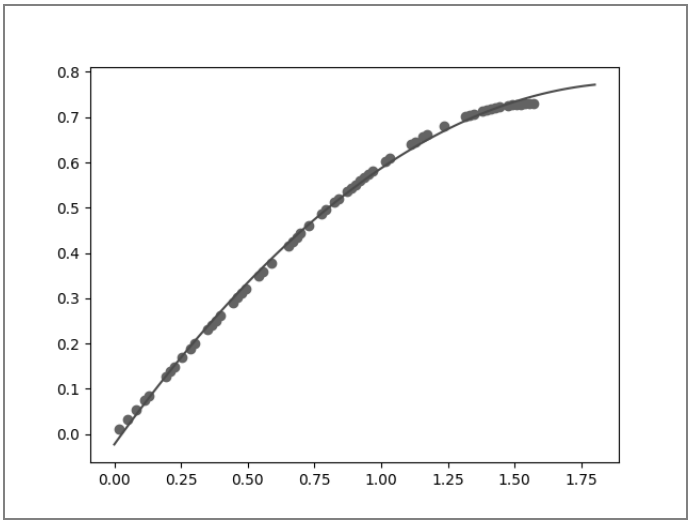


图 5-1-3　拟合为二次函数

果然拟合得很好，这些数据点就是符合二次函数模型的。

且慢下此结论。

在机器学习中，评价一个模型是否“好”是有量化指标的，直觉、观察都无法揭露深层次的关系。

方法之一就是利用“测试集”数据来检验。即计算 X_test 中每个数据所对应的预测值，记作 y，然后比较 y 和测试集中的 Y_test 数据，如果两者趋同的程度高，则说明这个模型比较“好”。

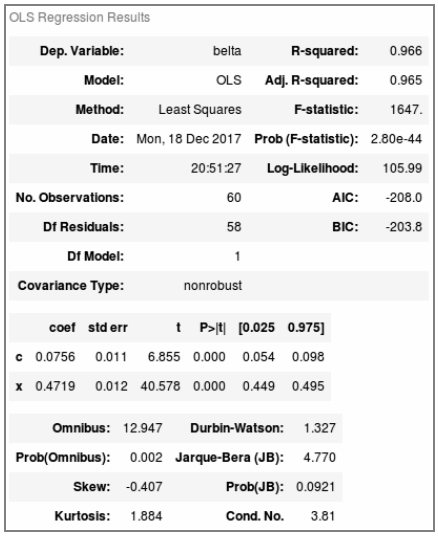
```
In [9]: coe = result.params
        y = coe['c'] + coe['x']*X_test
        np.corrcoef(y, Y_test)    #计算相关系数
Out[9]: array([[ 1.        ,  0.99252036],
               [ 0.99252036,  1.        ]])

In [10]: coe = result2.params
         y = coe['c'] + coe['x']*X_test + coe['x**2']*X_test*X_test
         np.corrcoef(y, Y_test)
Out[10]: array([[ 1.        ,  0.99929732],
                [ 0.99929732,  1.        ]])
```

np.corrcoef()是专门计算两个数据相关系数的函数。从输出结果中可以看出，通过模型 $y = -0.022 + 0.821x - 0.211x^2$ 得到的预测值与真实值之间相关系数更高，说明 result2 这个模型更“好”。

这仅仅是一种简单的方法。

不论是 result 还是 result2，作为模型对象，都有方法 summary()，用于显示模型摘要。

```
In [11]: result.summary()
Out[11]:（输出结果如图 5-1-4 所示）
```



OLS Regression Results

Dep. Variable:	belta	**R-squared:**	0.966
Model:	OLS	**Adj. R-squared:**	0.965
Method:	Least Squares	**F-statistic:**	1647.
Date:	Mon, 18 Dec 2017	**Prob (F-statistic):**	2.80e-44
Time:	20:51:27	**Log-Likelihood:**	105.99
No. Observations:	60	**AIC:**	-208.0
Df Residuals:	58	**BIC:**	-203.8
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
c	0.0756	0.011	6.855	0.000	0.054	0.098
x	0.4719	0.012	40.578	0.000	0.449	0.495

Omnibus:	12.947	**Durbin-Watson:**	1.327
Prob(Omnibus):	0.002	**Jarque-Bera (JB):**	4.770
Skew:	-0.407	**Prob(JB):**	0.0921
Kurtosis:	1.884	**Cond. No.**	3.81

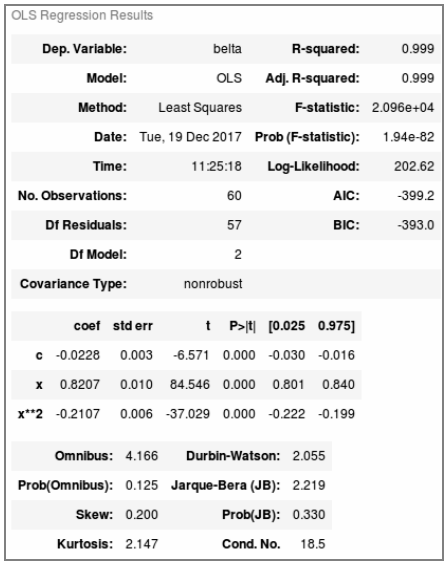
图 5-1-4　result 模型摘要

我们主要关注信息摘要中的如下内容。

- R-squared：判定系数，或者称为拟合度，其值范围是[0, 1]，值越大，表示模型拟合得越好。
- coef：显示的是 c 和 x 的系数，请参考 In[5]所创建的对象 X 进行理解。
- std err：相应系数的标准差。
- t 和 P：对每个系数的统计推断。原假设 H_0 认为系数不存在，P 值越小，则拒绝 H_0 的理由越充分，即该系数存在的理由越充分。

没有比较，无法定好坏。再来看看 result2.summary()。

```
In [12]: result2.summary()
Out[12]:（输出结果如图 5-1-5 所示）
```



OLS Regression Results

Dep. Variable:	belta	**R-squared:**	0.999
Model:	OLS	**Adj. R-squared:**	0.999
Method:	Least Squares	**F-statistic:**	2.096e+04
Date:	Tue, 19 Dec 2017	**Prob (F-statistic):**	1.94e-82
Time:	11:25:18	**Log-Likelihood:**	202.62
No. Observations:	60	**AIC:**	-399.2
Df Residuals:	57	**BIC:**	-393.0
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
c	-0.0228	0.003	-6.571	0.000	-0.030	-0.016
x	0.8207	0.010	84.546	0.000	0.801	0.840
x2**	-0.2107	0.006	-37.029	0.000	-0.222	-0.199

Omnibus:	4.166	**Durbin-Watson:**	2.055
Prob(Omnibus):	0.125	**Jarque-Bera (JB):**	2.219
Skew:	0.200	**Prob(JB):**	0.330
Kurtosis:	2.147	**Cond. No.**	18.5

图 5-1-5　result2 模型摘要

从 R-squared 的值可以看出，后面的模型要优于前面的模型。

```
In [13]: print(result.rsquared, result2.rsquared)
Out[13]: 0.965974637844  0.998641952133
```

这是简单获得 R-squared 值的方法。

既然比较的结果是第二个模型好，那么就把它作为反映数据关系的规律，可否？

断断不可！

因为还有一个重要的指标没有检验，那就是“残差”。

已知自变量的数据集为 X（由若干个 x 值组成），利用第二个模型 $y = -0.022 + 0.821x - 0.211x^2$ 可以计算出每个 x 所对应的结果，记作 y_estimate（并且组成一个因变量数据集）。因为这个值是利用上述模型计算得到的，而模型是我们依据假设拟合出来的，还不能确定作为“准确值”，只能称之为 “预测值”。此外，x 还对应着一个已经测得的真实值，记作 y_true，真实值和预测值之间的差，就叫作残差（英文为 Residual）——可以想一下，如果残差为 0，则说明那个模型准确地反映了数据中所隐含的规律，它就是“真理”。

在前面显示了线性回归的一般表达式，考虑到“误差”的存在——“误差”在各种测量中是不可绝对避免的——更通用的函数式为 $Y = \beta_0 + \beta_1 X + \epsilon$，其中 ϵ 被称为“误差”——严格地说是“随机误差”（Random Error）。这种误差无法控制，并且服从统计学上所谓的“正态分布”，随机性和不可预测性是 ϵ 的特征。

$\beta_0 + \beta_1 X$ 部分是关于自变量 X 的函数，其中包含了回归模型中所有可解释、可预测的信息，具有确定性。

所以，一个回归模型可以认为是由“确定部分”和“随机部分”两部分组成的。

对比刚刚引入的“残差”和“误差”这两个概念，不难发现，数据集中残差分布越接近“正态分布”，那么数据模型就越接近“真理”。

于是，我们就要检验残差是否呈正态分布了。

可以通过绘制 Q-Q 图直观地进行观察。

```
In [14]: import statsmodels.graphics.api as smg
         fig, ax = plt.subplots()
         smg.qqplot(result2.resid, ax=ax)
         fig.tight_layout()
Out[14]:（输出结果如图 5-1-6 所示）
```

对于服从正态分布的数据，在图中会呈现斜率为 1 的正比例函数，否则，该数据分布就不服从原假设。图 5-1-6 很明显，不是很好的斜率为 1 的函数，所以直观地判定本残差数据集不完全符合正态分布。

为了更有力地进行论证，继续绘制“残差图”，深入研究。

```
In [15]: import seaborn as sns
         sns.residplot(x=X_train.values, y=result2.resid.values)
Out[15]:（输出结果如图 5-1-7 所示）
```

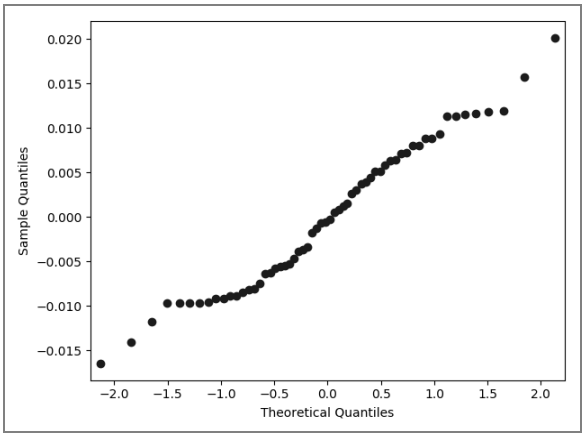


图 5-1-6　残差的 Q-Q 图

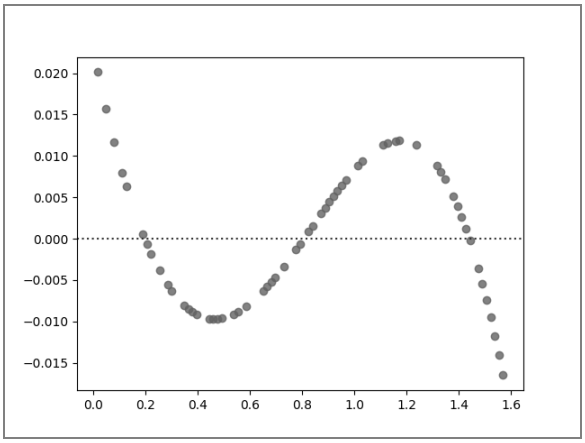


图 5-1-7　残差图

残差图的纵坐标是残差，横坐标是 X_train 的数据。从图 5-1-7 中可以明显地看出，残差是可预测的，即说明其中有一部分规律没有被纳入到“确定部分”的表达式，也说明函数模型没有完全反映数据的规律。

如何把残差中的规律完全筛检出来？

读者可以继续尝试增加高阶项的方法，假设函数模型为 $y = b + a_0x + a_1x^2 + a_2x^3$，甚至还可以创建更高阶的模型。

笔者可以很负责地预言，依然不能把残差中的规律完全筛检出来。

为什么如此自信？

这要从很早的时候说起，对于类似本例中的数据，从古希腊人托勒密（公元 2 世纪）开始，历经众多学者研究，直到 1621 年斯涅尔才终于找到了其中的规律，这就是伟大的斯涅尔定律（也叫做光的折射定律），即 $n = \dfrac{\sin\alpha}{\sin\beta}$，$n$ 是常数，为物质折射率。

所以，用 $Y = \beta_0 + \beta_1 X + \cdots + \epsilon$ 模式，不论做多少高阶的项，也无法发现其中的奥秘。

虽然我们折腾半天，回归结果也不成功，但是通过这个过程，读者已经了解了基本的线性

回归方法，特别是如何检验线性回归结果模型。

同时，笔者还在表明一种观点，机器学习，乃至于人工智能，是因“人工”而“智能”，是人而不是机器来管理世界。

无论如何，线性回归都是一种基本的、常用的回归分析方法，它的具体模式除 $Y=\beta_0+\beta_1 X+\epsilon$ 外，还可以是 $Y=\beta_0+\beta_1 X+\beta_2 X^2+\beta_3 X^3+\cdots+\epsilon$，或者是 $Y=\beta_0+\beta_1\sin(X)+\beta_2\cos(X)+\epsilon$ 等。

请注意，要判断是否为线性回归，不是看 Y 与变量 X 之间的关系，而是看 Y 与系数 β_i 之间是否为线性函数关系。例如 $Y=\exp(\beta_0+\beta_1 X)$，$Y$ 与系数 β_i 就不是线性关系，这种模型也不称之为线性回归模型，而是称之为非线性回归模型。某些非线性回归模型，可以通过变量变换转换为线性回归模型。例如刚才这个非线性模型，就可以分别用两个线性模型表示：$Y=\beta_0+\beta_1 X$，$\tilde{Y}=\log Y$。这样，就能够用通常的线性模型来处理非线性问题了。

3．scikit-learn 的工具

关于“线性回归”的工具，除 statsmodels 外，著名的机器学习库 scikit-learn 也提供了非常好的工具——LinearRegression。

（1）普通最小二乘法

前面在研究斯涅尔定律的时候，使用的就是最小二乘法，这种方法简单、实用，在 scikit-learn 中也有工具实现此方法，并且操作简单。

姑且先造一些数据，供演示所需。

```
In [16]: err = np.random.RandomState(1)
         x = 10 * err.rand(200)
         y = 2 * x - 5 + err.randn(200)
         x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.4,
                                                             random_state=40)
```

In[16]编造了两个数据集 x 和 y，虽然读者已经从代码中看到了 x 中的每个数值和 y 之间的对应关系，但先假装不知道——一定要统一口径，假装不知道函数关系，否则就演砸了。

然后还是用老办法，将数据集分为训练集和测试集两部分。

```
In [17]: from sklearn.linear_model import LinearRegression    #①
         model = LinearRegression(fit_intercept=True)    #②
         model.fit(x_train[:, np.newaxis], y_train)    #③
Out[17]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

先逐行理解 In[17]的含义，即熟悉 scikit-learn 的使用方法。

根据我们对训练集 x_train 和 y_train 的特点的了解——注意要假装不知道它们的关系——和我们丰富的专业知识（可以画出散点图观察，此处略），假设它们符合线性模型。于是就可以使用 sklearn 中的线性模型 LinearRegression——①引入。

依据丰富的 Python 经验，LinearRegression 是一个类——对类的理解，请参阅《跟老齐学 Python：轻松入门》。②其实创建了一个线性回归模型的实例对象。这个实例对象应该具有一些属性和方法，其中 fit()就是方法之一，它的作用是接收传入的数据来“训练”模型，即通过训练集数据，学得模型中的有关系数，从而拟合得到线性回归的数学模型——这就是③。

LinearRegression 这个模型的具体使用方法，可以通过文档获得。

```
In [18]: LinearRegression?
Init signature: LinearRegression(fit_intercept=True, normalize=False, copy_X=True,
n_jobs=1)
Docstring:
Ordinary least squares Linear Regression.
#省略余下内容
```

对文档的内容，这里简单描述，仅供参考。

- LinearRegression()：是用 OLS 方法来完成线性回归的。
- fit_intercept：是否计算常数项（截距），默认为 True；如果为 False，则不计算。
- normalize：默认为 False。如果 fit_intercept=False，则此参数被忽略；如果为 True，则训练集数据在回归之前将被归一化。
- copy_X：默认为 True，此时训练集数据会被复制；若为 False，则会把原训练集数据覆盖。
- n_jobs：默认为 1，计算需要的 CPU 数量；如果为–1，则使用所有可用的 CPU。

阅读了上面的说明，再看 In[17]中②所创建的线性模型实例，就不难理解了。

为了更直观地看到这个数据模型的函数样式，可以使用模型对象的两个属性。

- coef_：以数组形式返回变量的系数。
- intercept_：以数组形式返回截距。

```
In [19]: print("slope: ", model.coef_[0])
         print("intercept: ", model.intercept_)
Out[19]: slope:  1.99154872449
         intercept:  -4.88227590023
```

从 Out[19]可知，通过 In[17]的③训练得到的线性模型的函数式是 $y = 1.991x - 4.882$（因为随机划分数据集，所以读者得到的结果可能与此稍有差异，但不影响后续判断），再回头看看 In[16]中创建这个数据集时所使用的函数表达式，两者非常接近——因为 In[16]中加入了干扰因素。

后续步骤应该是用这个模型进行预测，使用测试集中的 x_test 数据。

```
In [20]: y_fit = model.predict(x_test[:, np.newaxis])     #①
         np.corrcoef(y_fit, y_test)
Out[20]: array([[ 1.        ,  0.98499746],
                [ 0.98499746,  1.        ]])
```

In[20]的①使用模型对象的 predict()方法，传入了测试集数据，得到了预测数据 y_fit，然后计算预测数据 y_fit 和测试集中的 y_test（真实值）之间的相关系数。从 Out[20]的结果可以看出，两者相关系数比较高，说明预测得比较准确，即模型有效。

但是，我们在 In[10]中也做过类似的相关系数计算，曾经得到了同样的结论，结果呢？结果是我们也没有发现正确的规律。要判断是否正确，还需要对残差进行分析。

下面绘制残差图。

```
In [21]: resid = y_test - y_fit
```

```
        sns.residplot(x=x_test, y=resid)
```
Out[21]:（输出结果如图 5-1-8 所示）

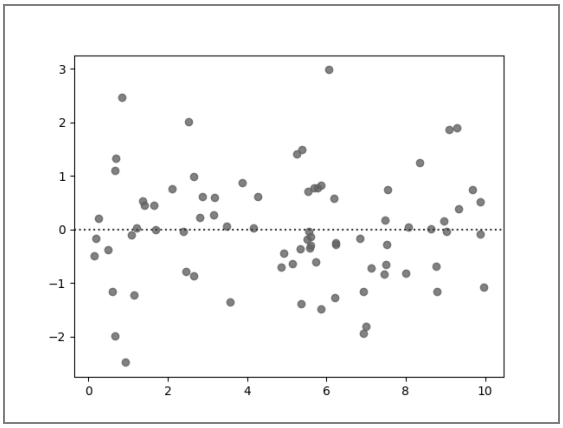


图 5-1-8　残差图

这里的残差显然是随机的、不可预测的，读者可以比较图 5-1-7 和图 5-1-8 两张残差图。

可以非常肯定地说，In[17]的③所训练出来的模型完全反映了数据之间的规律，即 $y = 1.991x - 4.882$ 就是数据集 x 和 y 之间的“真理”。

貌似操作流程很简单。

但笔者建议，凡是阅读到此处的读者，如果是机器学习的初学者，请暂停，并重新阅读前述内容，特别要理解基本流程。如有必要，自己绘制流程图，以辅助理解——心急吃不了热豆腐。

（2）多项式函数

按照前面所示的方式使用 sklearn.linear_model.LinearRegression，所得的线性回归模型默认是一次函数（$y = ax + b$），如果要得到多项式函数，怎么办？可以使用 klearn.preprocessing.PolynomialFeatures 来创建多项式函数。

官方文档显示多项式类的完整形式是 PolynomialFeatures(degree=2, interaction_only=False, include_bias=True)，即通过它可以创建一个多项式实例。

- degree：整数，默认是 2，表示多项式最高阶的次数。
- interaction_only：默认为 False。如果为 True，则变量会产生交叉。例如，打算创建 $y = b_0 + b_1x_1 + b_2x_2 + b_3x_1x_2$ 多项式，通常我们提供的数据集包括 x_1 和 x_2，在使用 PolynomialFeatures 创建多项式实例的时候，不需要在数据集中计算 x_1*x_2，而是直接在其参数中声明 interaction_only=True，则交叉项在模型中自动创建。
- include_bias：默认为 True。通常线性函数表达式中都要有常数项（截距），默认所创建的多项式模型也有常数项，并且是 1，然后通过训练得到其系数。如果 include_bias = False，则截距为 0。

下面看一个例子，理解上面的说明。

```
In [22]: from sklearn.preprocessing import PolynomialFeatures
         poly = PolynomialFeatures(3, include_bias=True)
```

poly 所引用的对象就是一个多项式实例，请读者对照上面的参数说明，理解这个多项式实例的特点。

然后，我们向这个多项式实例提供一组数据，看看结果是否如自己所愿。

```
In [23]: x = np.array([2, 3, 4])
         poly.fit_transform(x[:, None])
Out[23]: array([[  1.,   2.,   4.,   8.],
                [  1.,   3.,   9.,  27.],
                [  1.,   4.,  16.,  64.]])
```

面对输出结果，读者是否理解了多项式实例 poly 的作用？请参考图 5-1-9。

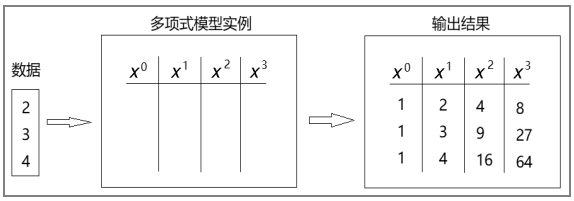


图 5-1-9　多项式模型的理解

再看一个有交叉项的例子。

```
In [24]: poly = PolynomialFeatures(2, interaction_only=True)
         x = pd.DataFrame({"x1": [1, 2, 3], "x2": [4, 5, 6]})
         poly.fit_transform(x)
Out[24]: array([[  1.,   1.,   4.,   4.],
                [  1.,   2.,   5.,  10.],
                [  1.,   3.,   6.,  18.]])
```

从 Out[24]的结果不难理解 interaction_only=True 导致的结果。

理解多项式函数的创建方法之后，下面看一个假想的例子——数据还是自己造的。

```
In [25]: err = np.random.RandomState(1)
         x = 10 * err.rand(100)
         y = np.sin(x) + 0.1 * err.randn(100)
         x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.4,
random_state=40)
```

我们假装已经有了训练集和测试集的数据，又假装通过散点图发现分布是曲线——你假装不知道它们符合正弦函数模型。创建多少次的多项式函数模型呢？本来应该根据经验和屡次尝试来最终确定可能的函数最高次数，但是因为我们毕竟知道数据之间是正弦关系，所以直接把多项式次数设置得高一些，比如 7 次。

```
In [26]: from sklearn.pipeline import Pipeline    #①
         model = Pipeline([('poly', PolynomialFeatures(degree=7)),
                           ('linear', LinearRegression(fit_intercept=False))])   #②

         model = model.fit(x_train[:, np.newaxis], y_train)
         y_fit = model.predict(x_test[:, np.newaxis])

         np.corrcoef(y_fit, y_test)
Out[26]: array([[ 1.        ,  0.98994052],
```

```
       [ 0.98994052,  1.        ]])
```

在 In[26]中使用了 Pipeline 类，与之等效的还有一个名为 make_pipeline 的函数（引入方式：from sklearn.pipeline import make_pipeline），它们的目标都是将若干个“转换器”（Transformer）与“评估器”（Estimator）连接起来。

在②中，PolynomialFeatures 就是一个转换器——转换器的作用是根据指定的多项式形式（具体看参数设置）重新组合输入的变量。对于 In[26]中的②而言，就是依据变量创建了一元七次多项式。这个多项式创建完毕，就把结果输入到 LinearRegression——评估器。这样，最终我们通过 Pipeline 类实现了一个以多项式函数为基础的线性回归模型。

后面的步骤就不稀奇了。

看最后的结果，y_fit 和 f_test 很吻合。

当然，读者可以用前述方法检验残差，从而确定拟合结果是否正确。此处省略了，不过，可以用一个特殊值看看。

```
In [27]: import math
         math.sin(math.pi/2)
Out[27]: 1.0

In [28]: model.predict([[math.pi/2]])
Out[28]: array([ 1.03096889])
```

还比较满意。

为什么本来是正弦函数，改用多项式也能拟合得很好呢？笔者觉得读者是能够回答这个问题的。

LinearRegression 是 scikit-learn 中线性回归的基本工具，此外还有其他模型，但是本书就不过多介绍了——因为笔者还要写一本专门的关于“机器学习”的书，还要留点东西在那里用呢。

（3）理解 Pipeline

在 In[26]中使用了 scikit-learn 中的 Pipeline 类（或 make_pipeline 方法），虽然已经做了简要说明，但根据笔者经验，对于初学者而言，它依然是一个迷一样的存在，所以有必要重申。

最权威的说明当然来自官方文档（http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html），但其内容多少有点晦涩，所以就有了如下解释。

文档中显示其参数列表样式是 Pipeline(steps, memory=None)，这里的 steps 代表了什么？

语言是思维的工具，程序员的语言就是代码，所以要通过代码说明（以下内容参考了两篇文章：*Using scikit-learn Pipelines and FeatureUnions*，网址是 http://zacstewart.com/2014/08/05/pipelines-of-featureunions-of-pipelines.html；*scikit-learn: Pipeline*，网址是 http://yyao.info/scikit-learn/2016/ 09/15/scikit-learn-pipeline。特向两篇文章的作者表示感谢）。

```
from sklearn.base import TransformerMixin
from sklearn.base import BaseEstimator
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline

class FooTransformer(TransformerMixin):
```

```
    def __init__(self, number):
        self.number = number

    def fit(self, X, y=None):
        print("Transformer No.{}: fit X={}".format(self.number, X))
        return self

    def transform(self, X):
        print("Transformer No.{}: transform X={} => X+10={}".format(self.number, X,
              X+10))
        return X+10

class BarEstimator(BaseEstimator):
    def __init__(self, number):
        self.number = number

    def fit(self, X, y=None):
        print("Estimator No.{}: fit X={}".format(self.number, X))
        return self

    def predict(self, X):
        print("Estimator No.{}: predict X={} is...".format(self.number, X))
        return "a new result"
```

在 scikit-learn 中，提供了转换器、评估器的基础类，即上述代码中所引入的 TransformerMixin 和 BaseEstimator。在机器学习中，对数据进行一些预处理，比如降维、正则化等，所使用的工具我们称之为“转换器”，它的经典方法就是 fit()和 transform()；而类似于 LinearRegression 这样的数学模型即为“评估器”，它的经典方法则是 fit()和 predict()。scikit-learn 中的转换器和评估器，都要继承刚才所说的两个基础类。

上述代码中定义了简单的转换器类 FooTransformer 和评估器类 BarEstimator，目的在于理解 Pipeline 的工作流程。请注意观察两个类的方法的异同。

先来看看 Pipeline 类和 make_pipeline()方法的异同。

```
if __name__ == '__main__':
    p1 = Pipeline(steps=[("Trans1", FooTransformer(1)),
                         ("Trans2", FooTransformer(2)),
                         ("Estmt1", BarEstimator(1))])
    print(p1)
    p2 = make_pipeline(FooTransformer(1), FooTransformer(2), BarEstimator(1))
    print(p2)
#输出结果如下
#Pipeline(memory=None,
#         steps=[('Trans1', <__main__.FooTransformer object at 0x7f2c95c0de48>),
('Trans2', #<__main__.FooTransformer object at 0x7f2c95c0de80>), ('Estmt1',
BarEstimator(number=1))])
#Pipeline(memory=None,
#         steps=[('footransformer-1', <__main__.FooTransformer object at
0x7f2c95c52c18>), #('footransformer-2', <__main__.FooTransformer object at
0x7f2c9718de48>), ('barestimator', #BarEstimator(number=1))])
```

毋庸多言，只需仔细观察两者的使用方法和输出结果，就能比较容易地理解其异同了，所以，不再赘述。重点看“转换器”和“评估器”的使用方法。

```
if __name__ == '__main__':
    p = Pipeline(steps=[("Trans1", FooTransformer(1)),
                        ("Trans2", FooTransformer(2)),
                        ("Estmt1", BarEstimator(1))])
    print("#===== Pipeline fitting =====#")
    p.fit(X=100)
    print("#===== Pipeline predicting =====#")
    pred = p.predict(X=100)
    print(pred)

#输出结果：
##===== Pipeline fitting =====#
#Transformer No.1: fit X=100
#Transformer No.1: transform X=100 => X+10=110
#Transformer No.2: fit X=110
#Transformer No.2: transform X=110 => X+10=120
#Estimator No.1: fit X=120
##===== Pipeline predicting =====#
#Transformer No.1: transform X=100 => X+10=110
#Transformer No.2: transform X=110 => X+10=120
#Estimator No.1: predict X=120 is...
#a new result
```

Pipeline 类中的参数 steps 所引用的是一个列表，其中以元组为元素，其结构为（”names”，“instance”），每个元组包含转换器名称和实例对象。注意，不论前面有多少个转换器，最后一组必须是评估器。

例如上述执行代码，p.fit(X=100)的效果是按照顺序依次执行每个转换器中的 fit()和 transform()方法，并将数据依次传递。到最后的评估器时，数据是经过前述各个转换器修正过的，最后执行评估器的 fit()方法。p.predict(X=100)不执行转换器中的 fit()方法，只执行 transform()方法，也将最终结果数据传入到评估器，并在最后执行评估器的 predict()方法。

上述内容简述了 Pipeline 的工作过程，还请读者注意，它的作用只是“管道”，而不是某种特定的语法规则。

如果读者感觉上述解释仍是浅尝辄止，建议深入研究 scikit-learn 这个库，或者等待笔者后续关于机器学习的书籍。

5.2　线性回归示例

5.1 节中关于斯涅尔定律的研究，可以看作线性回归的例子，但是，那个例子中的数据有点特殊，其实不是笔者通过实验测得的数据，是笔者编造的，并且在编造过程中，为了演示需要，没有引入数据的偶然误差。

本节要演示的示例，都是来自真实世界的数据。

1．铁路客运量

提到“春运”，读者立刻就会联想到火车票不好买、人多。之所以有这种感觉，可能是因为大部分人都集中在那几天出行。如果到国家数据（http://data.stats.gov.cn）网站上查看有关数据，或许会有新的发现。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt

        pg = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter05/train_
                         passengers.csv")
        pg['date'] = pd.to_datetime(pg['date'])

        fig, ax = plt.subplots()
        ax.plot(pg.date, pg.number, color='red', marker="o", markerfacecolor='b')
        ax.xaxis_date()
Out[1]:（输出结果如图 5-2-1 所示）
```

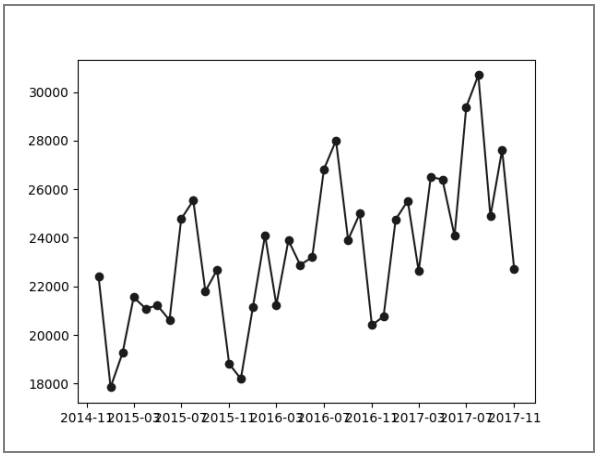


图 5-2-1　每月运量图

从图 5-2-1 所示的折线图中可以大致看出每个月旅客运量的变化。当然，随着铁路建设和人民生活水平的提高，总体运量在不断提高，铁路承载着我们的中国梦——这段最符合主流价值观了，恳请转发。

程序员的价值观，不仅仅局限在中国梦，还有美国梦，可以称之为“世界大同梦”吧。为了实现这个梦、那个梦，就要把数据分析再深入——为上述数据建立数学模型。

很显然，可以假设是多项式模型。

```
In [2]: from sklearn.preprocessing import PolynomialFeatures
        from sklearn.linear_model import Ridge     #①
        from sklearn.pipeline import make_pipeline

        pg['date'] = pd.to_datetime(pg['date'], unit='s')    #②
        pg['stamp'] = [t.value//10**9 for t in pg.date]     #③
```

```
        model = make_pipeline(PolynomialFeatures(degree=9), Ridge())   #④
        model.fit(pg.stamp[:, np.newaxis], pg['number'])    #⑤
        pg_pred = model.predict(pg.stamp[:, np.newaxis])     #⑥

        np.corrcoef(pg_pred, pg['number'])     #⑦
Out[2]: array([[ 1.        ,  0.64958439],
               [ 0.64958439,  1.        ]])
```

In[2]除引入我们熟悉的 PolynomialFeatures 外，还在①中引入了 Ridge，这其实是另外一个模型 Ridge Regression，常翻译为“岭回归”——此处不讲述这种模型，请期待笔者后续的“机器学习”拙作——再宣传就有点啰嗦了。

②和③是对时间列进行适当处理，以时间戳来表示时间列。因为在④所创建的模型中，自变量应该是连续值。

⑤是利用数据训练模型，⑥是应用此模型评估同样自变量数据集所得到的因变量，然后在⑦中对预测数据和真实数据相互比较，得到相关系数。从输出结果看，这个模型还是差强人意的——商务印书馆《现代汉语词典》：“【差强人意】大体上还能使人满意。”——尽管误差有点大。

```
In [3]: fig, ax = plt.subplots()
        ax.plot(pg.stamp, pg['number'], color='red', marker="o",
                markerfacecolor='r', label="passengers")
        ax.plot(pg.stamp, pg_pred, color='blue', label="line of model")
        ax.legend()
Out[3]:（输出结果如图 5-2-2 所示）
```

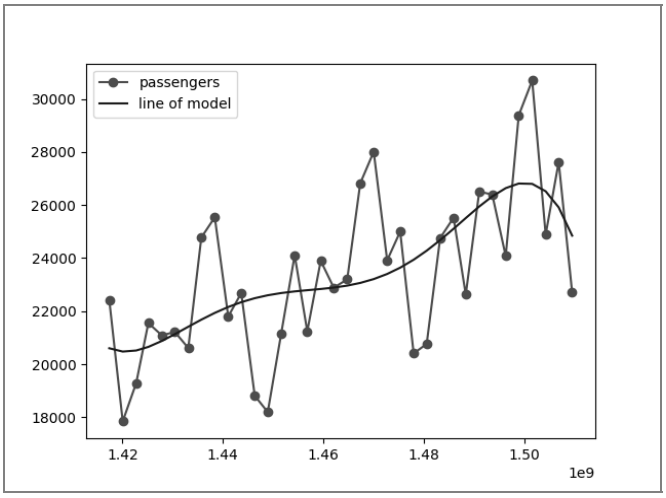


图 5-2-2　运量模型

In[3]是把所得数据模型用图示的方式表示出来，并与原来的折线图做比较。如果用这个折线图预测一下后续铁路运量，会不会准呢？读者可以尝试一番——如果仅仅是预测某个月的，估计不会准。

不过，我们的目的不在于预测是否准，而在于是否会使用学过的工具，毕竟面对铁路旅客运输这种复杂事件，仅用上面的工具预测，的确简陋了一些。

2．蓝鳃太阳鱼的尺寸

蓝鳃太阳鱼产于北美，因鳃盖边缘有浅蓝色而得名。笔者在一门在线的课程（https://onlinecourses.science.psu.edu/stat501/）中找到了一些数据，有研究者（Cook and Weisberg, 1999）专门测量了这种鱼的年龄和长度——已经把数据保存到本书的代码仓库中，需要的读者可以下载。

```
In [4]: bluegills = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter05/
                                bluegills.txt", sep="\t")
        plt.scatter(bluegills['age'], bluegills['length'])
Out[4]:（输出结果如图 5-2-3 所示）
```

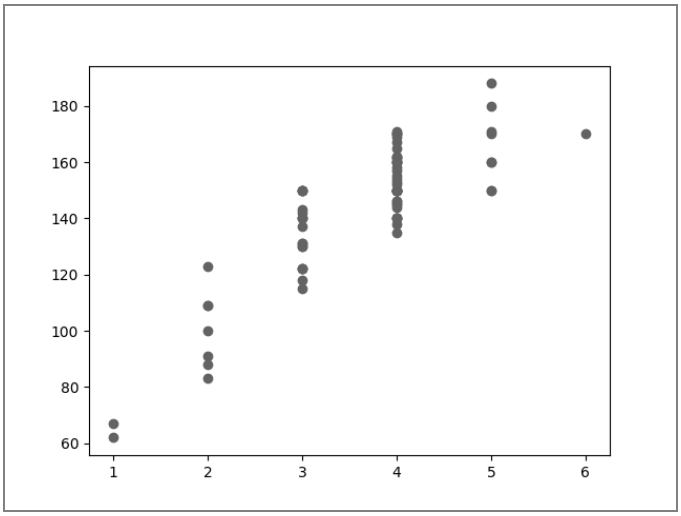


图 5-2-3　年龄与长度数据

下面要做的就是将上述数据拟合为曲线，找出年龄和长度之间的规律。

```
In [5]: from sklearn.preprocessing import PolynomialFeatures
        from sklearn.linear_model import LinearRegression

        bluegills = pd.read_csv("/home/qiwsir/Documents/DataAnalysis/chapter05/
                                bluegills.txt", sep="\t")

        feature_2 = PolynomialFeatures(degree=2)    #①
        age_2 = feature_2.fit_transform(bluegills['age'].reshape(-1, 1))    #②

        regressor_2 = LinearRegression(fit_intercept=False)    #③
        regressor_2.fit(age_2, bluegills['length'])    #④

        year = bluegills.age.unique()
        year_2 = feature_2.transform(year[:, None])    #⑤

        length_pred = regressor_2.predict(year_2)    #⑥

        fig, ax = plt.subplots()
        ax.scatter(bluegills.age, bluegills.length, color='red')
```

```
        ax.plot(year, length_pred, color='blue')

        regressor_2.coef_    #⑦
Out[5]: array([ 13.62237616,  54.04931191,  -4.71866479])
```

（输出结果如图 5-2-4 所示）

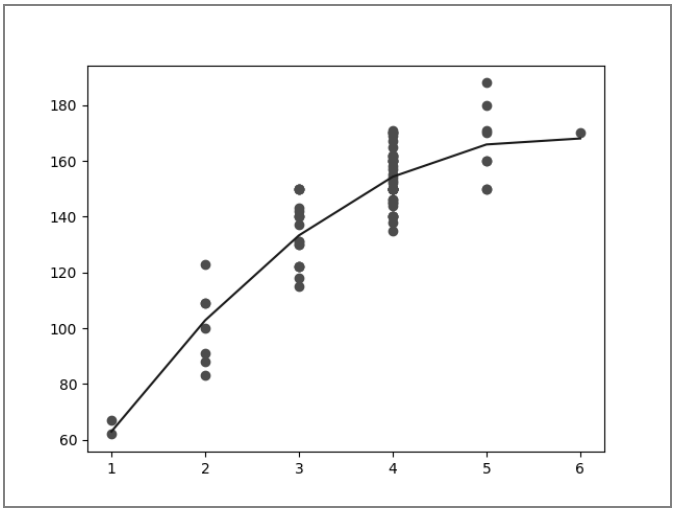


图 5-2-4　年龄与长度拟合为曲线

In[5]的代码与前面的代码稍有不同，但这种不同仅仅是表面上的，其思想仍然一致。①仍然是创建一个最高次为 2 次的多项式，然后用②来创建以年龄为变量的这个多项式模型。凭借我们的经验和专业知识，还是要进行线性拟合——这也是仅有的工具，③创建线性回归模型（注意参数），并用④训练此模型。这里没有使用 Pipeline 类，但从①到④的操作，也是要将数据用多项式对象进行转换，然后传入到线性回归模型 LinearRegrassion 中。

请注意对比②和⑤，②其实执行的是 fit()和 transform()两个方法，⑤仅执行了 transform()方法。

经过⑤所转换的数据仍然传入到评估器⑥，并执行其 predict()方法。

特别建议读者结合本示例，阅读 5.1 节中“理解 Pipeline”的内容。

通过⑦得到此二次多项式各项的系数。当然，前面先画出图像来了。

由此，我们可知，拟合出来的二次多项式是：$y = 13.622 + 54.049x - 4.719x^2$。对照 https://onlinecourses.science.psu.edu/stat501/node/325 中给出的结果，完全吻合。

要检验这个模型的正确度或准确度，就要画出残差 Q-Q 图。

```
In [6]: import statsmodels.graphics.api as smg
        fig, ax = plt.subplots()
        length_pred2 = regressor_2.predict(feature_2.transform(bluegills.age[:,
                                           np.newaxis]))
        resid = length_pred2 - bluegills.length
        smg.qqplot(resid, ax=ax)
Out[6]:（输出结果如图 5-2-5 所示）
```

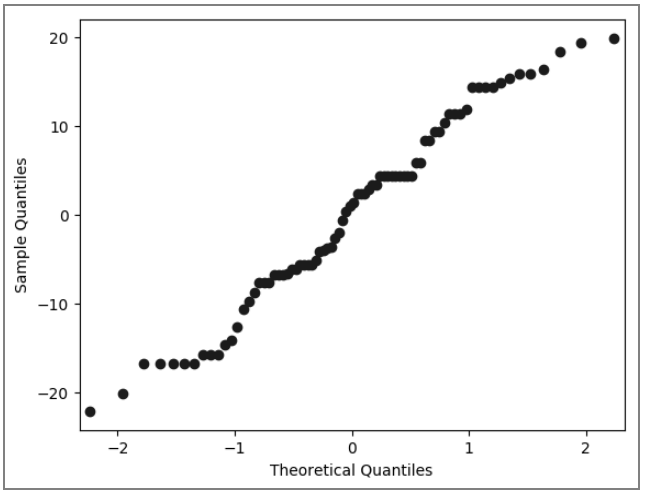


图 5-2-5　残差 Q-Q 图

从图 5-2-5 看，所得到的二次函数模型可以接受。

线性回归是一种简单且常用的回归模型，围绕着这种模型，还有很多拓展工具，比如已经遇到过的岭回归等，想深入学习的读者，敬请期待后续的精彩。

5.3　Logistic 回归

常言道，“物以类聚，人以群分”。在人类看来，给这个世界进行“分类”是很重要的。阿猫阿狗本来不知道自己是哪个阵营的，我们非要划分出不同的门、纲、目、科、属、种，结果猫儿和狗儿发现对方“非我族类，必有异心”，于是破坏了和谐。生物学中就有专门干这个事情的学科——生物分类学。

无独有偶，机器学习中也有“分类”——它是一种数学模型，本节将要介绍的 Logistic Regression 即为其一，虽然它的名字中含有“Regression”，但却不同于前面的“线性回归”中的“回归”。

对于 Logistic Regression（也称为 Logit Regression）的中文译名，通常资料里面翻译为“逻辑回归”。南京大学的周志华教授翻译为“对数几率回归”（详见周志华所著《机器学习》）。现在就要开始选边站队了，读者站在哪一边？——这就是分类，而且是最简单的分类，通常称之为“二元分类”，要么赞成周教授，要么反对，没有第三条道路可选。

本书在以中文名称说明 Logistic 回归时，采用“对数几率回归”的译名，有时候也会直接使用 Logistic，不进行翻译，这算是第三条道路了。

什么是 Logistic 回归（对数几率回归）呢？稍安勿躁。要说清楚它，还需要从“线性回归”开始。

线性回归模型中的因变量（预测值）通常是连续的。在现实中，也会对连续值进行分类。比如依据大学录取分数把考生分为“名落孙山”和“金榜题名”两类，而从能力角度看，很难区分录取线上下 1 分范围的考生。所以，这种有点武断又无奈的做法如今备受指责。

还有一种连续值，直观地看会有相对的集中现象。比如，在火车站，可以把人群的分布划

分为两类，即“上车”和“候车”（暂不考虑车站外面的，当然，这是简化模型）。

那么，如何把连续值变成离散值呢？

聪明的人类发明了一门名为“数学”的东西，数学就提供了一个聪明的函数：

$$y=\frac{1}{1+\mathrm{e}^{-z}}$$

这就是著名的 Sigmoid 函数。使用下面的方法可以画出这个函数的图像。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from scipy.stats import logistic

        fig = plt.figure(figsize=(12,8))
        ax = fig.add_subplot(111)
        z = np.linspace(-6, 6, 1000)
        ax.plot(z, logistic.cdf(z), 'r-', label='Logistic')
Out[1]:（输出结果如图 5-3-1 所示）
```

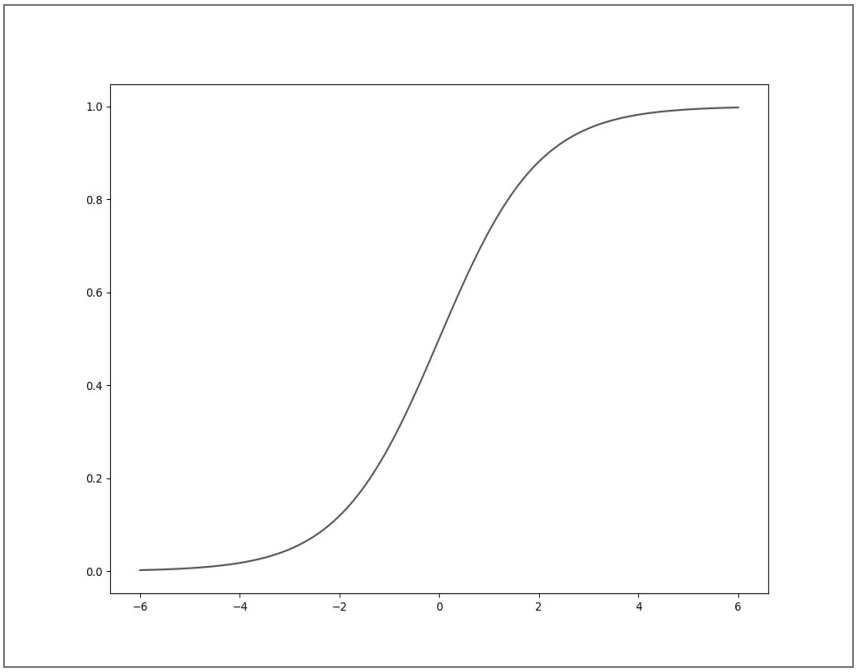


图 5-3-1　Sigmoid 函数图像

通过图像不难发现，当 z=0 时，y=0.5，这个位置是临界点。z 值越大，y 越趋近于 1；z 值越小，y 越趋近于 0，并且两个方向的趋近速度也很快。这样，我们就可以借助 z 把所研究的对象划分为两类，一类标记为 1，另一类标记为 0。

假设，z 与所观测的自变量 X 之间存在着 $z=\boldsymbol{\beta}^{\mathrm{T}}X+b$ 的关系，然后就可以根据 Sigmoid 函数得到回归结果 y。注意 y 是连续值，但根据图 5-3-1 可知，y 很容易趋近于 0 或 1，那么就可以

进一步设置条件，如果 y>0.5，则 y=1；如果 y<0.5，则 y=0（y=0.5 时，可以根据需要设置归属）。

但是，$z=\boldsymbol{\beta}^{\mathrm{T}}X+b$ 是假设的，所以要通过计算得到这个表达式，即确定系数 β 和常数 b。一旦这二者确定，表达式 $z=\boldsymbol{\beta}^{\mathrm{T}}X+b$ 就得到了，然后可以按照上述流程操作，最终实现对结果的二元分类。这样看来，确定 β、b 是关键。

至于如何确定 β 和 b，数学上常用“极大似然法”，但这里不介绍了，请读者参考有关资料。

显然，对数几率回归的结果是将对象分成两类——二元分类是一种简单且常用的分类方法，比如：好人/坏人、敌人/朋友、墙内/墙外、阴/阳、黑/白，等等。

接下来以一个示例，帮助读者初步理解“对数几率回归”的操作过程。

1．分析鸢尾花的类型

在很多数据分析工具中，都集成了鸢尾花（Iris）的数据（前面章节已经用过了），下面从刚刚熟悉不久的 statsmodels 中获得其数据集。

```
In [2]: import statsmodels.api as sm
        df = sm.datasets.get_rdataset("iris").data
        df.info()
Out[2]: <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 150 entries, 0 to 149
        Data columns (total 5 columns):
        Sepal.Length    150 non-null float64
        Sepal.Width     150 non-null float64
        Petal.Length    150 non-null float64
        Petal.Width     150 non-null float64
        Species        150 non-null object
        dtypes: float64(4), object(1)
        memory usage: 5.9+ KB
```

通过 df.info()能够看到此数据集的摘要，共计 150 条记录，字段分别为花瓣的长度（Petal.Length）、花瓣的宽度（Petal.Width）、花萼的长度（Sepal.Length）、花萼的宽度（Sepal.Width）和花的类型（Species）。

再看看有几种类型。

```
In [3]: df['Species'].unique()
Out[3]: array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

有三种类型的数据不便于我们演示“对数几率回归”，我们所需要的数据中只要有两种类型即可，所以用下面的方式筛选出后面要用到的数据集。

```
In [4]: iris = df[(df.Species == "versicolor") | (df.Species == "virginica")]
        iris['Species'] = iris['Species'].map({'versicolor': 1, 'virginica': 0})
        iris.rename(columns = {"Sepal.Length": 'sepal_length',
                               "Sepal.Width": 'sepal_width',
                               "Petal.Length": "petal_length",
                               "Petal.Width": "petal_width"},
                    inplace=True)
        iris.head()
Out[4]:
        sepal_length      sepal_width  petal_length petal_width  Species
```

```
50        7.0          3.2          4.7          1.4          1
51        6.4          3.2          4.5          1.5          1
52        6.9          3.1          4.9          1.5          1
53        5.5          2.3          4.0          1.3          1
54        6.5          2.8          4.6          1.5          1
```

使用 In[4]的代码对原有数据集进行了筛选，并且将有关字段名称予以修改，特别是修改了 Species 的值，分别用 1 和 0 代表原来两种花的类型名称，这样做的目的就是将类型结果和 Sigmoid 函数的结果对应起来。

```
In [5]: from statsmodels.formula.api import logit    #①
        model = logit("Species ~ petal_length + petal_width", data=iris)    #②
        result = model.fit()

        from scipy import stats     #③
        stats.chisqprob = lambda chisq, df: stats.chi2.sf(chisq, df)    #④
        result.summary()   #⑤
Out[5]:
        Optimization terminated successfully.    #本行开始以下三行并未显示在 Out[5]中，但为了
            Current function value: 0.102818        #与代码行区分开，在本书中放置于此
        Iterations 10
```

（输出结果如图 5-3-2 所示）

Logit Regression Results

Dep. Variable:	Species	**No. Observations:**	100
Model:	Logit	**Df Residuals:**	97
Method:	MLE	**Df Model:**	2
Date:	Fri, 29 Dec 2017	**Pseudo R-squ.:**	0.8517
Time:	22:03:18	**Log-Likelihood:**	-10.282
converged:	True	**LL-Null:**	-69.315
		LLR p-value:	2.303e-26

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	45.2723	13.612	3.326	0.001	18.594	71.951
petal_length	-5.7545	2.306	-2.496	0.013	-10.274	-1.235
petal_width	-10.4467	3.756	-2.782	0.005	-17.808	-3.086

图 5-3-2　输出结果

在 statsmodels 库中有实现对数几率回归的方法，①引入了 logit 方法，在②中使用它创建一个对数几率回归模型的实例对象。当然，根据经验，也会有名为 Logit 的类，可以用 sm.Logit()形式来使用，效果与②一样。

②中的“Species ~ petal_length + petal_width”表示多项式 $z = \boldsymbol{\beta}^{\mathrm{T}}X + b$ 在本示例中的具体形式是 $z = \beta_0 + \beta_1\text{petal\_length} + \beta_2\text{petal\_width}$，后面的任务就是要找出 β_0、β_1、β_2 三个系数。

②中的 data=iris 表明了此模型所用的数据集，之后使用模型的 fit()方法进行训练。

如此这般，就建立了 Logistic 回归模型实例。在使用它的 predict()方法之前，要先对这个模

型有一个比较全面的了解。模型实例的 summary()方法可以显示关于此模型的摘要信息，如⑤的操作所示。

不过，这里增加了③和④，它们不是必需的，只因笔者在调试的时候报错了，才用这两句代码来修正。如果读者直接执行⑤不报错，就不用执行③、④两句。

从⑤的输出结果可以看出，所创建的模型还是相当不错的（请运用数理统计知识查看输出结果的各项指标）。

由 Out[5]输出结果中的 coef 值，可以写出本例中的多项表达式 $z = 45.27 - 5.75\text{petal\_length} - 10.44\text{petal\_width}$，这就是我们要寻找的表达式。将这个多项式代入 Sigmoid 函数式，即可得到：

$$\text{Species} = \frac{1}{1+e-(45.27-5.75\text{petal\_length}-10.44\text{petal\_width})}$$

如果此时有某一组数据，其中包含了鸢尾花的花瓣长度和宽度，根据上面的表达式就可以预测其类型了。

当然，在这里我们没有必要用上面的计算过程，因为已经得到了一个训练好的 Logistic 回归模型对象。这个对象有 predict()方法，我们要做的就是把花瓣的数据传给这个方法，让它来给每朵花归类。

```
In [6]: new_iris = pd.DataFrame({"petal_length": np.random.randn(100)*0.5+5,
                                 "petal_width": np.random.randn(100)*0.5+1.7})
        new_iris.head()
Out[6]:
           petal_length  petal_width
        0     5.548834     1.771221
        1     5.567582     2.251131
        2     4.712078     1.265089
        3     5.629662     1.721155
        4     4.844296     1.735620
```

In[6]新创建了一个数据集，其中包括了鸢尾花的花瓣长度和宽度两个字段，然后用刚刚得到的 Logistic 回归模型对这个数据集中的鸢尾花进行分类。

```
In [7]: new_iris['pre_species'] = result.predict(new_iris)
        new_iris.head()
Out[7]:
            petal_length  petal_width  pre_species
        0      5.548834     1.771221     0.005698
        1      5.567582     2.251131     0.000034
        2      4.712078     1.265089     0.992900
        3      5.629662     1.721155     0.006035
        4      4.844296     1.735620     0.323921
```

可以看到，得到的并不是 0 和 1。那当然，通过 Sigmoid 函数得到的也是连续值，只是因为它比较快速地收缩到 0 和 1，我们可以把 pre_species 的值更好地归类到 1 和 0。

```
In [8]: new_iris["Species"] = (new_iris['pre_species'] > 0.5).astype(int)
        new_iris.head()
Out[8]:
          petal_length   petal_width  pre_species  Species
```

```
0    5.548834      1.771221     0.005698       0
1    5.567582      2.251131     0.000034       0
2    4.712078      1.265089     0.992900       1
3    5.629662      1.721155     0.006035       0
4    4.844296      1.735620     0.323921       0
```

至此，我们根据最初的 iris 数据集训练得到了一个对数几率回归模型，并用此模型对新的数据集 new_iris 中的数据进行了分类。

为了更直观地反映这种分类，还可以在坐标系中，按照（petal_length, petal_width）的坐标点对每个数据描点，从而能够观察到这些数据的分布特点。

已知 $$z = \beta_0 + \beta_1 \text{petal\_length} + \beta_2 \text{petal\_width}$$

令 $$z = 0$$

得 $$\text{petal\_width} = -\frac{\beta_0}{\beta_2} - \frac{\beta_2}{\beta_2}\text{petal\_length}$$

在坐标系中画出这条线，表示不同类别鸢尾花的分布界限。

```
In [9]: p = result.params
        alpha0 = -p['Intercept']/p['petal_width']
        alpha1 = -p['petal_length']/p['petal_width']

        fig, ax = plt.subplots()
        ax.plot(iris[iris['Species']==0].petal_length.values,
                iris[iris['Species']==0].petal_width.values, 's',
                label='virginica')
        ax.plot(new_iris[new_iris['Species']==0].petal_length.values,
                new_iris[new_iris['Species']==0].petal_width.values,
                'o', markersize=10, color='steelblue',
                label='virginica(pred)')
        ax.plot(iris[iris['Species']==1].petal_length.values,
                iris[iris['Species']==1].petal_width.values,
                's',
                label='versicolor')
        ax.plot(new_iris[new_iris['Species']==1].petal_length.values,
                new_iris[new_iris['Species']==1].petal_width.values,
                'o', markersize=10, color='green',
                label='versicolor(pred)')

        x = np.linspace(3.5, 7, 20)
        ax.plot(x, alpha0 + alpha1*x, "k")
        ax.set_xlabel("Width")
        ax.set_ylabel("Length")
        ax.legend()
Out[9]:（输出结果如图 5-3-3 所示）
```

上述代码量虽然不少，但比较容易理解，请读者耐心阅读。

至此，我们已经初步领略了使用 statsmodels 中的方法完成 Logistic 回归的基本过程。

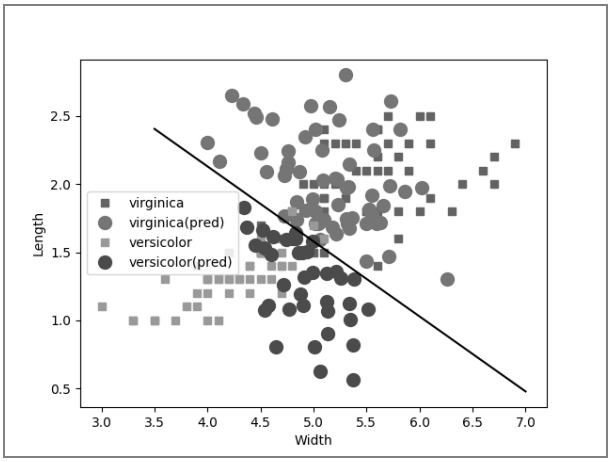


图 5-3-3　鸢尾花分类

2．再分析泰坦尼克数据

之前曾经分析过泰坦尼克数据，只不过前面的分析是基于统计学的分析，而此处再次分析，则是要依据此数据建立 Logistic 回归模型。

```
In [10]: url = 'https://raw.githubusercontent.com/BigDataGal/
                                Python-for-Data-Science/master/titanic-train.csv'
         titanic = pd.read_csv(url)
         titanic.columns = ['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age',
                            'SibSp', 'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked']
         titanic.head()
Out[10]:（输出结果如图 5-3-4 所示）
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

图 5-3-4　输出结果

重点要关注的字段是“Survived”，它代表该乘客是否获救，其值有且只有两类——获救（用 1 表示）和遇难（用 0 表示）。所以，本例就要研究乘客获救的数学模型。

在处理真实的数据项目时，要对数据集的情况有足够多的了解，然后清洗数据，最后才是使用某些工具进行分析。

```
In [11]: titanic.isnull().sum()
Out[11]: PassengerId      0
         Survived         0
         Pclass           0
```

```
Name             0
Sex              0
Age              177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin            687
Embarked         2
dtype: int64
```

每个乘客都被标记了是否获救，但 Age、Cabin 和 Embarked 三个字段有缺失数据现象，不过不用担心，后面我们就要把它处理掉。

```
In [12]: titanic.info()
         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 891 entries, 0 to 890
         Data columns (total 12 columns):
         PassengerId     891 non-null int64
         Survived        891 non-null int64
         Pclass          891 non-null int64
         Name           891 non-null object
         Sex             891 non-null object
         Age             714 non-null float64
         SibSp            891 non-null int64
         Parch           891 non-null int64
         Ticket           891 non-null object
         Fare            891 non-null float64
         Cabin           204 non-null object
         Embarked       889 non-null object
         dtypes: float64(2), int64(5), object(5)
         memory usage: 83.6+ KB
```

数据集 titanic 中共有 891 条记录。

对数据集有了初步了解之后，就开始“清洗”，最简单的清洗就是“消灭”，凭主观判断，没用的东西“消灭”后数据集就干净了——抬头可以看到天际线。

```
In [13]: titanic_data = titanic.drop(['PassengerId','Name','Ticket','Cabin'], 1)
```

“消灭”一些暂时用不到的字段——当然，这是根据个人经验删除的。比如，在我们的经验中，遇难者是否获救与其姓名无关，于是可以将 Name 列删除。其他亦然。

注意，Age 字段没有被“消灭”，因为是否获救跟年龄肯定有关——妇女儿童优先。但是，前面已经显示，这个字段中有很多缺失数据，要对其进行处理。

```
In [13]: def age_approx(cols):
         Age = cols[0]
         Pclass = cols[1]

         if pd.isnull(Age):
             if Pclass == 1:
                 return 37
             elif Pclass == 2:
```

```
                return 29
            else:
                return 24
        else:
            return Age

        titanic_data['Age'] = titanic_data[['Age', 'Pclass']].apply(age_approx, axis=1)
        titanic_data.dropna(inplace=True)
        titanic_data.isnull().sum()
Out[13]: Survived    0
        Pclass      0
        Sex         0
        Age         0
        SibSp       0
        Parch       0
        Fare        0
        Embarked    0
        dtype: int64
```

这样处理之后，解决了缺失数据问题。不过，还有一些字段的数据不便于用来分析，需要进行进一步整理。

```
In [14]: gender = pd.get_dummies(titanic_data['Sex'], drop_first=True)
        embark_location = pd.get_dummies(titanic_data['Embarked'], drop_first=True)
        titanic_data.drop(['Sex', 'Embarked'], axis=1, inplace=True)
        titanic_reg = pd.concat([titanic_data, gender, embark_location], axis=1)
        titanic_reg.head()
Out[14]:
```

	Survived	Pclass	Age	SibSp	Parch	Fare	male	Q	S
0	0	3	22.0	1	0	7.2500	1	0	1
1	1	1	38.0	1	0	71.2833	0	0	0
2	1	3	26.0	0	0	7.9250	0	0	1
3	1	1	35.0	1	0	53.1000	0	0	1
4	0	3	35.0	0	0	8.0500	1	0	1

尽可能用可计算的数值来做各个字段的值，便于将其作为变量参与运算。

虽然已经删除了一些和建立最终模型无关的字段，但是仅凭经验和直觉，还不足以清除干净，比较理智的做法是画出各变量（字段视为变量）的相关系数“热图”。

```
In [15]: import seaborn as sb
        sb.heatmap(titanic_reg.corr())
Out[15]:（输出结果如图 5-3-5 所示）
```

从图 5-3-5 中可以直观地看到各个变量（字段）之间的相关性。对于 Logistic 回归而言，其变量必须是独立变量——不仅是此回归，机器学习中都有此要求。据此，显然 Fare（船票价格）和 Pclass（船舱等级）之间肯定是有因果关系的（图 5-3-5 中也显示了它们之间有很强的负相关性）。所以，必须删除它们。

```
In [16]: titanic_reg.drop(['Fare', 'Pclass'],axis=1,inplace=True)
        titanic_reg.head()
Out[16]:
```

Survived	Age	SibSp	Parch	male	Q	S

```
0         0   22.0      1      0      1    0    1
1         1   38.0      1      0      0    0    0
2         1   26.0      0      0      0    0    1
3         1   35.0      1      0      0    0    1
4         0   35.0      0      0      1    0    1
```

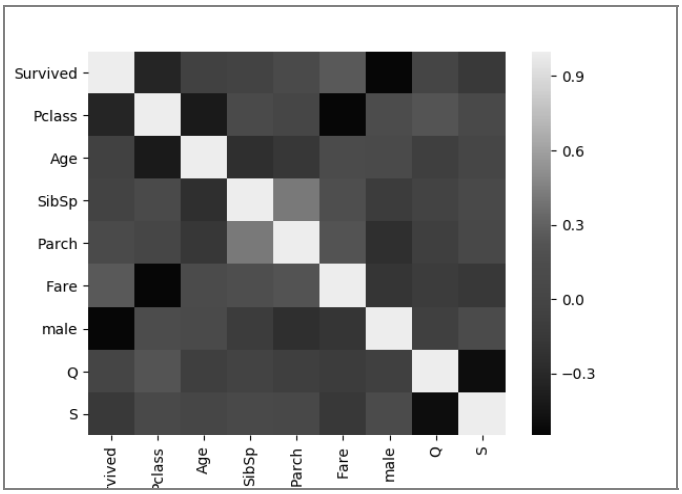


图 5-3-5　不同字段的关系热图

至此，数据清洗工作完毕。

接下来，就利用 scikit-learn 中的工具创建一个 Logistic 回归模型，这个模型的因变量是 Survived，自变量是 titanic_reg 数据集中的其他字段。

```
In [17]: y = titanic_reg.iloc[:, 0].values     #①
         X = titanic_reg.iloc[:, 1:6].values    #②

         from sklearn.cross_validation import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4,
                                                             random_state=25)   #③
```

In[17]的①和②分别把因变量和自变量数据集从 titanic_reg 中分离出来，然后在③中分别创建训练集和测试集。

```
In [18]: from sklearn.linear_model import LogisticRegression
         logit_model = LogisticRegression()
         logit_model.fit(X_train, y_train)
Out[18]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                            intercept_scaling=1, max_iter=100, multi_class='ovr',
                            n_jobs=1, penalty='l2', random_state=None,
                            solver='liblinear', tol=0.0001, verbose=0,
                            warm_start=False)
```

在 In[18]中使用的是 sklearn 中的 LogisticRegression 创建对数几率回归模型实例，然后用训练集数据训练此模型。

得到模型实例之后，就可以用它做出预测了，当然，使用的是测试集数据。

```
In [19]: y_pred = logit_model.predict(X_test)
```

y_pred 就是用模型预测的乘客是否获救的结果，与真实的结果 y_test 相比，准确率有多高呢？

```
In [20]: pd.crosstab(y_test, y_pred, rownames=['True'], colnames=['Predicted'],
                     margins=True)
Out[20]:
         Predicted      0    1   All
              True
                 0    182   30   212
                 1     46   98   144
               All    228  128   356
```

Out[20]显示了预测结果和真实结果的对比，自我感觉还不错吧，虽然有一些误差——人命关天，最好别有误差，差一个就是一条命呀。

类似的检验还可以通过下面的方法进行，效果是一样的。

```
In [21]: from sklearn.metrics import confusion_matrix
         confusion_matrix = confusion_matrix(y_test, y_pred)
         confusion_matrix
Out[21]: array([[182,  30],
                [ 46,  98]])
```

在我们所建立的 Logistic 回归模型中，没有像“分析鸢尾花的类型”示例那样编写多项式的表达式，而是直接使用 sklearn 方法创建模型实例，把相关的数学问题交给工具完成。

以上仅仅是以示例的方式，帮读者简要了解对数几率回归的使用方法。在真正的机器学习项目中，通常要比这里演示的更复杂。所以，还要请读者关注后续专门讲述机器学习的拙作。

Logistic Regression 或 Logit Regression 是应用非常广泛的工具，在社会学、生物统计学、临床、数量心理学、计量经济学、市场营销等统计实证分析中都可以看到它的身影。

5.4 贝叶斯方法

上一节使用 Logistic 回归解决的是“二元分类”问题，然而世界是复杂的，仅用“阴、阳”难以解释万物。所以，还要探索新的分类方法。在进入到本节主题内容之前，不妨用数学语言把分类问题进行适度抽象表达——抽象的才能更普适。

首先要有待分类对象，用 X 表示待分类对象的集合，即 $X=\{x_1, x_2, ..., x_n\}$，其中每个元素都可以被看作是一个待分类对象，集合 X 可称之为“项集合”。与“项集合”相应的是“类别集合”，用 Y 表示，即 $Y=\{y_1, y_2, ..., y_m\}$，其中每个元素表示一个类别。我们需要做的事情就是找到某种对应关系，让 X 中的元素与 Y 中的元素对应，即 $y_i = f(x_i)$，且只有一个 y_i。那么我们就把这种对应关系（函数 f）称为“分类器”。

在分类问题中，我们的任务就是寻找适合的分类器。

如何构建分类器呢？

在面对一些作为训练集的数据时，构造适合的分类器，是知识和经验的融合过程，也是尝试和选择的过程。在实践中，不要苛求分类器 100%地解决分类问题，但可以让它工作得越来越好。

朴素贝叶斯就是诸多分类器中一个简单、易用、效果好的分类器。它是以著名的贝叶斯定

理为基础的。

1．贝叶斯定理

$$P(\mathrm{H}|\mathrm{D})=\frac{P(\mathrm{H})P(\mathrm{D}|\mathrm{H})}{P(\mathrm{D})}$$

这就是著名的贝叶斯定理。

关于贝叶斯发现这个定理的故事，坊间有很多猜测，因为未见诸于正史，可以权当八卦看看。但不论如何，这个定理火了。作为一名机器学习工程师，如果不知道贝叶斯定理，就可以用“数典忘祖”这个成语来形容了。

这个定理是什么意思？

很抱歉，笔者不在这里做完整的数学阐述。因为笔者在这本书中主要是打算通过示例，让走进数据分析大门的读者也窥视一下机器学习。关于机器学习的完整阐述，请期待后续拙作推出。

不过，要用数学语言来说明这个贝叶斯定理在分类问题上的应用。

假设有一个待分类对象 $x = \{a_1, a_2, ..., a_n\}$，其中 a_i 是这个对象的特征属性（比如 x 是一个 DataFrame 类型的对象，a_i 则为其列名称）。类别集合依然用 $Y = \{y_1, y_2, ..., y_m\}$。然后就要用某种方法——这个方法就是“分类器”——判定 x 到底是属于 y_1，还是属于 y_2，还是 Y 中的其他类别。为此，采用一种概率估计的方法，分别计算 $P(y_1|x)$、$P(y_2|x)$、... $P(y_m|x)$的概率——将这种概率命名为“后验概率”。然后找这些概率中最大的，比如是 $P(y_k|x)$，那么我们就认定 x 属于 y_k 这个类。

如何计算 $P(y_i|x)$呢，这就要用到贝叶斯定理了。

$$P(y_i|x)=\frac{P(x|y_i)P(y_i)}{P(x)}$$

根据有关数学知识（如果在此处感觉数学知识不够用，一定要去补充）可知：

- $P(x)$与 Y 无关，在这个式子中可以看作常数；
- $P(y_i)$是一个比较好计算的量。

于是，计算 $P(y_i|x)$的问题转换为计算 $P(x|y_i)$——这是解决问题的重要思路，通过各种变换推动问题不断转换，从而找到解决问题的突破口。

$P(x|y_i)$就容易计算了吗？是的，比较起来确实容易计算了——请用数学知识来判断。

回到我们假设的待分类对象 x，它有若干个特征属性，即 $x = \{a_1, a_2, ..., a_n\}$。

为了计算 $P(x|y_i)$，针对 x 做了一个大胆的假设：各个特征属性相互完全独立。这就是所谓“条件独立”假设，这个假设有人为的痕迹，但在贝叶斯定理使用范围中依然能够得到令人满意的结果，并且计算量还减少了。

因为条件独立假设是如此“朴素”——直接、简单，所以在这个条件下应用贝叶斯定理得到的那个分类器又叫作“朴素贝叶斯分类器”。

2．朴素贝叶斯分类器

接前面所述，依据条件独立假设来计算 $P(x|y_i)$，即：

$$P(x|y_i)=P(a_1,a_2,\cdots,a_n|y_i)=P(a_1|y_i)P(a_2|y_i)\ldots P(a_n|y_i)$$

如此，就得到一个重要的关系：

$$P(y_i|x)\propto P(x|y_i)P(y_i)$$

然后找出概率最大的，那么 x 就属于该类别——最大似然（概率）。

完成上述整个计算，直到给出最终结果，这个过程所用到的函数（分类器）就是朴素贝叶斯分类器。特别注意“条件独立假设”这个条件。

可以说，朴素贝叶斯分类器具有简单、直接、高效的特点。

为了便于理解，下面引述一个司空见惯的例子：垃圾邮件分类。

如何确定一封邮件是否为垃圾邮件呢？

有一封邮件，用 E 来表示。此邮件由 N 个单词构成，那么这些单词就是此邮件的特征属性（默认邮件是英文的，如果是中文的，则需要先分词），特征量集合可以表示为 $\{d_1, d_2, .. d_N\}$。

如果是垃圾邮件，则标记为 S。根据贝叶斯定律，可以将此邮件作为垃圾邮件的概率表示为：

$$P(\mathrm{S}|\mathrm{E})=\frac{P(\mathrm{S})P(\mathrm{E}|\mathrm{S})}{P(\mathrm{E})}$$

$P(\mathrm{S})$表示垃圾邮件的概率，只需要计算一个邮件库中垃圾邮件占总邮件的比例即可。

要计算 $P(\mathrm{E}|\mathrm{S})$，按照朴素贝叶斯分类器所要求的，E 的各个特征属性（单词）之间是相互独立的关系——这个假设其实很暴力，后面我们会看到一种被称为“向量化”的处理技巧。尽管很暴力，但从结果来看，这个假设还是可以接受的，我们就能够使用朴素贝叶斯分类器来处理此问题。从严格的逻辑上看，上述说明陷入了“循环论证”的漩涡，其实数学上有对此的严格证明，此处从略，所以可以这么计算：

$$P(\mathrm{E}|\mathrm{S})=P(d_1|\mathrm{S})P(d_2|\mathrm{S})\ldots P(d_N|\mathrm{S})$$

$P(d_i|\mathrm{S})$，表示单词 d_i 在垃圾邮件中出现的频率，这个值可以通过训练集的数据获得。

于是就得到了 $P(\mathrm{S}|\mathrm{E})$，从而得知该邮件是垃圾邮件的概率。

在具体应用朴素贝叶斯分类器的时候，我们没有必要按照上面演示的流程去编程，因为 scikit-learn 提供了很好的工具，而且不止一种，不同的模型适用于不同场景。

3．高斯模型

高斯模型是 scikit-learn 提供的一种朴素贝叶斯分类器，适用于待分类对象的特征属性是连续值的情况。这里不展示数学公式了，直接通过代码示例讲解具体应用过程。

```
In [1]: %matplotlib
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
```

```
        from sklearn import datasets

        iris = datasets.load_iris()
        iris.feature_names
Out[1]: ['sepal length (cm)',
         'sepal width (cm)',
         'petal length (cm)',
         'petal width (cm)']
```

在示例中，还是使用我们熟悉的鸢尾花数据，但是这次数据来自于 sklearn。Out[1]显示了此数据集具有 4 个特征属性。

```
In [2]: iris.target_names
Out[2]: array(['setosa', 'versicolor', 'virginica'],
              dtype='<U10')
```

Out[2]显示了鸢尾花的三个类别，跟曾经用过的数据集一样。不过要注意，这里的 iris 已不是以往简单的 DataFrame 对象了，“它变了”，变成了一个类字典对象。读者如果使用过类似 MongoDB 这样的 NoSQL 数据库，对这个对象的结构理解就会更深刻。

```
In [3]: iris.target
Out[3]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
               1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
               1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
               2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
               2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

iris.target 的值标记了每种花的类别，并且已经把类别数值化了，0 代表 setosa，其他依此类推。

依然要划分训练集和测试集，这是机器学习的惯例。

```
In [4]: from sklearn.cross_validation import train_test_split
        iris_data_train, iris_data_test, iris_target_train, iris_target_test =
                                train_test_split(iris.data, iris.target, test_size = 0.3,
                                                 random_state=25)
```

完成数据准备之后，就要应用高斯模型了。

```
In [5]: from sklearn.naive_bayes import GaussianNB
        iris_model = GaussianNB()
        iris_model.fit(iris_data_train, iris_target_train)
Out[5]: GaussianNB(priors=None)
```

引入 GaussianNB 类，创建实例化对象，然后利用训练集的数据对这个模型进行训练——按照惯例完成规定动作。

模型训练完毕，也就是分类器已经构造好了，下面就要用它对测试集的数据进行分类。

```
In [6]: result = iris_model.predict(iris_data_test)
        np.corrcoef(result, iris_target_test)
Out[6]: array([[ 1.        ,  0.94894007],
               [ 0.94894007,  1.        ]])
```

把预测数据和测试数据进行比较，会发现预测的结果是相当不错的。请读者将其与前述的

Logistic 回归模型比较一番。

从 In[4]到 In[6]，仅仅几步，就完成了朴素贝叶斯分类器之一——高斯模型的应用。代码如此简练，皆因 sklearn 给我们提供了“肩膀”。

在这个示例中，我们没有做任何前置分析，上来就使用高斯模型。如果按照严格的逻辑，应该先确认待分类对象是否符合高斯模型的应用条件，即特征服从正态分布。如果不满足此条件，分类器就不好用了，然后再找其他的分类器，直到找到分类效果相对满意的为止——在实际中，我们常常用结果来确定原因。

4．多项式模型

多项式模型，也是朴素贝叶斯分类器的一个具体实现，其名称为 MultinomialNB（Multinomial Naive Bayes）。

多项式模型适用于离散值，比如文本分类（请温习“识别垃圾邮件”的示例）。接下来，我们使用 scikit-learn 提供的一组新闻数据，演示多项式模型的应用方法，当然，中间会有其他知识补充。

```
In [7]: from sklearn.datasets import fetch_20newsgroups
        news = fetch_20newsgroups(subset='all')
        print(news.keys())
Out[7]: dict_keys(['data', 'filenames', 'target_names', 'target', 'DESCR',
                   'description'])
```

这个数据集还是类字典对象，注意 key 跟前面鸢尾花的数据集稍有差别。

如果用 news.data 查看，得到的是各个记录的文本内容，每条文本内容被归类到某个类别中（news.target_names 显示的类别）。

```
In [8]: print(news.data[35])
        print(news.target[35])

        From: UC525655@mizzou1.missouri.edu (M.Eaton)
        Subject: (Q) Way to connect PB 145, IIsi, P LW LS?
        Nntp-Posting-Host: mizzou1.missouri.edu
        Organization: University of Missouri
        Lines: 7

        Is there a way to connect a PowerBook 145, Mac IIsi, and Personal LaserWriter
        LS so that I can (not necessarily silmultaneoulsy) print from either the IIsi,
        or PB, and file share between the IIsi and PB? I know I can get the ($expensive$)
        LW NT upgrade for my LS, but I can't afford that...

        Thanks, Mark

        4
```

In[8]显示了 news.data[35]的文本内容及相应的类别编号（news.target[35]的值，即 4），然后用 In[9]可以知道类别名称。

```
In [9]: news.target_names[3]
Out[9]: 'comp.sys.ibm.pc.hardware'
```

下面要做的是划分训练集和测试集。

```
In [10]: news_train = fetch_20newsgroups(subset="train", remove=('headers', 'footers',
                                         'quotes'))
         news_test = fetch_20newsgroups(subset='test', remove=('headers', 'footers',
                                         'quotes'))
```

读者也可以继续使用 In[4]的方式划分，不过 In[10]也值得尝试。在 In[10]中，使用了数据集本身的参数，不仅可以确定训练集和测试集，还能去除文档中无关的内容，这样我们得到的就是每个文档的内容部分了。

以训练集为例，news_train.data 里面的内容是文本，要用于多项式模型中，需要对文本内容进行一番处理，将文本内容转换成数值形式的特征向量，这也是用任何机器学习算法处理文本所必须要做的工作。向量化最直接的做法是用“词袋”来表示。

什么是“词袋”？先看一个很简单的例子。

比如有两个文本内容：

- 文本一，Learn Python with Laoqi. Laoqi is a Python programmer.
- 文本二，Laoqi write some books of Python.

把这两个文本中出现的不重复单词（大小写忽略）挑出来，可以组成一个列表：['learn', 'python', 'with', 'laoqi', 'is', 'a', 'programmer', 'write', 'some', 'books', 'of']。

将文本内容和列表进行比较，以列表为类比对象，依次标记文本中每个单词相对列表中的单词出现的次数。两个文本的标记结果分别是：

- 文本一，[1, 2, 1, 2, 1, 1, 1, 0, 0, 0, 0]
- 文本二，[0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1]

如此，我们其实可以用一个矩阵来表征单词出现的次数（严格地讲，标点符号等也应该纳入进来，这里暂且简化处理），矩阵的每一行对应一个文本，每一列表示一个不重复的单词在不同文本中出现的次数。

经过处理之后的文本，就转换成了词袋（Bag of Words）。用术语来说明，词袋模型是在自然语言处理和信息检索下被简化的表达模型。在这种模型中，文本被看作是无序的词汇集合，忽略语法甚至是单词的顺序。因此，它满足了贝叶斯定理中的条件独立性假设。

接下来，我们要做的就是根据训练集数据，学得一个分类器，这个分类器能够将测试集中的文本数据进行归类，即属于 news.target_names 中的哪一类。此处的分类器自然要用多项式模型，输入到这个模型的文本数据就要利用词袋模型进行转换，并以单词的出现次数作为特征属性。把文本数据转换为词袋模型，我们也称之为文本“矢量化（向量化）”。

继续使用巨人的肩膀，sklearn 中提供了文本矢量化的工具 CountVectorizer，完整的描述请参考官方文档（http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html），下面的示例仅供读者理解官方文档。

```
In [11]: from sklearn.feature_extraction.text import CountVectorizer
         count_vect = CountVectorizer(stop_words="english", decode_error='ignore')
         x_train = count_vect.fit_transform(news_train.data)
In [12]: x_train.toarray()
```

```
Out[12]: array([[0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                    ...,
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0]], dtype=int64)
```

Out[12]得到的结果就是 news_train.data 的文本中每个文本单词出现次数的矩阵——这就把文本矢量化了。诚然，我们在 In[11]的操作中，没有对文本做过多的限制，在真实的项目中，还可以对文本单词进行进一步筛选，从而剔除明显不在分析范围内的词汇。

然后，可以把这个矢量化的结果 x_train 传给 TfidfTransformer（from sklearn.feature_extraction.text import TfidfTransformer），它的作用是计算上述特征属性的权重（关于 TF-IDF，请读者关注一下）。在 sklearn 中有一个将上述两步合二为一的工具，下面就用这个工具完成本示例。

```
In [13]: from sklearn.feature_extraction.text import TfidfVectorizer
         from sklearn.naive_bayes import MultinomialNB
         from sklearn.pipeline import make_pipeline
         from sklearn import metrics

         pipe = make_pipeline(TfidfVectorizer(), MultinomialNB())
         pipe.fit(news_train.data, news_train.target)
         news_pred = pipe.predict(news_test.data)

         print(metrics.classification_report(
                   news_test.target, news_pred, target_names=news_test.target_names))
```

```
                          precision    recall  f1-score   support

         alt.atheism                0.81      0.07      0.13       319
         comp.graphics              0.72      0.62      0.67       389
         comp.os.ms-windows.misc    0.70      0.50      0.59       394
         comp.sys.ibm.pc.hardware   0.55      0.75      0.64       392
         comp.sys.mac.hardware      0.81      0.61      0.69       385
         comp.windows.x             0.83      0.74      0.78       395
         misc.forsale               0.86      0.69      0.77       390
         rec.autos                  0.82      0.68      0.74       396
         rec.motorcycles            0.89      0.63      0.73       398
         rec.sport.baseball         0.95      0.69      0.80       397
         rec.sport.hockey           0.59      0.90      0.71       399
         sci.crypt                  0.47      0.80      0.59       396
         sci.electronics            0.77      0.43      0.55       393
         sci.med                    0.86      0.63      0.73       396
         sci.space                  0.84      0.63      0.72       394
         soc.religion.christian     0.22      0.95      0.36       398
         talk.politics.guns         0.59      0.59      0.59       364
         talk.politics.mideast      0.85      0.70      0.77       376
         talk.politics.misc         0.81      0.08      0.15       310
         talk.religion.misc         0.50      0.00      0.01       251

         avg / total                0.72      0.61      0.61      7532
```

报告上显示了相对于测试集，预测的精确度和召回率等结果。

以上就是朴素贝叶斯分类器中多项式模型在文本分类中的基本应用流程。还是要提醒读者，上述示例不完全等同于真正的应用，在真正的应用中可能还需要做其他的操作，比如进行有关优化等，届时请读者根据实际情况来定，此示例只演示了基本的、必需的步骤。

5．伯努利模型

伯努利模型跟多项式模型一样，也是适用于离散数据的一个分类器，但是在数据特征值上，跟多项式模型不同。多项式模型中特征值可以是多个，但在伯努利模型中，只能是布尔值或者二进制值。虽然数值不再那么多样化了，但伯努利模型的名称霸气犹存——BernoulliNB，它的完整参数说明可以参考官方文档（http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html），以下先以简单示例说明基本用法。

```
In [14]: from sklearn.naive_bayes import BernoulliNB
         X = np.random.randint(2, size=(6, 100))
         Y = np.array([0, 1, 2, 3, 4, 5])
         clf = BernoulliNB()
         clf.fit(X, Y)
         print(clf.predict(X[2:3]))
Out[14]: [2]
```

这是来自官方文档中的示例，为了显示明确，对 *Y* 稍微进行了修改。

从示例中可以看出，用法和前面基本一样。只是注意，*X*（自变量）中的值是由 0、1 组成的。

从简单模型中不足以看出真正的应用，还要用近似于真实的东西。这次选择了著名的大数据竞赛网站 kaggle 上提供的数据——旧金山地区的犯罪数据（网址是 https://www.kaggle.com/c/sf-crime/data，读者可以在这里下载训练集和测试集数据，也可以到本书的代码仓库中下载）。

```
In [15]: crime_train = pd.read_csv('/Users/qiwsir/Documents/DataAnalysis/
                                chapter05/crime_train.csv',
                                parse_dates=['Dates'])
         crime_train.head()
Out[15]:（输出结果如图 5-4-1 所示）
```

	Dates	Category	Descript	DayOfWeek	PdDistrict	Resolution	Address	X	Y
0	2015-05-13 23:53:00	WARRANTS	WARRANT ARREST	Wednesday	NORTHERN	ARREST, BOOKED	OAK ST / LAGUNA ST	-122.425892	37.774599
1	2015-05-13 23:53:00	OTHER OFFENSES	TRAFFIC VIOLATION ARREST	Wednesday	NORTHERN	ARREST, BOOKED	OAK ST / LAGUNA ST	-122.425892	37.774599
2	2015-05-13 23:33:00	OTHER OFFENSES	TRAFFIC VIOLATION ARREST	Wednesday	NORTHERN	ARREST, BOOKED	VANNESS AV / GREENWICH ST	-122.424363	37.800414
3	2015-05-13 23:30:00	LARCENY/THEFT	GRAND THEFT FROM LOCKED AUTO	Wednesday	NORTHERN	NONE	1500 Block of LOMBARD ST	-122.426995	37.800873
4	2015-05-13 23:30:00	LARCENY/THEFT	GRAND THEFT FROM LOCKED AUTO	Wednesday	PARK	NONE	100 Block of BRODERICK ST	-122.438738	37.771541

图 5-4-1　输出结果

数据中显示了旧金山地区的犯罪记录，包括时间、地点及详细的经纬度坐标等。

下面整理数据，让数据中的特征值符合伯努利模型的要求。

```
In [16]: from sklearn import preprocessing
         le = preprocessing.LabelEncoder()
         crime = le.fit_transform(crime_train['Category'])
```

In[16]使用 LabelEncoder 类，用它来规范序列对象的值，通常是以对象中不重复元素组成的序列为基准，用如下简单示例进行说明，就更容易理解了。

```
In [17]: simple_le = preprocessing.LabelEncoder()
         simple_le.fit(["soochow", "shanghai", "soochow", "nanjing", "shanghai"])
         simple_le.classes_
Out[17]:array(['nanjing', 'shanghai', 'soochow'],
               dtype='<U8')
```

通过执行 LabelEncoder()实例的 fit()方法，实现了不重复元素的提取，然后用 transform()对["soochow", "shanghai", "soochow", "nanjing", "shanghai"]进行规范化。

```
In [18]: simple_le.transform(["soochow","shanghai","soochow","nanjing","shanghai"])
Out[18]: array([2, 1, 2, 0, 1])
```

观察 Out[18]和 Out[17]的输出内容，可知“规范”的结果是以 Out[17]中的索引来标记被规范对象。In[17]和 In[18]的过程可以合并为一个 fit_transform()方法来实现，这就是 In[16]的操作。

观察 Out[15]的结果，每条犯罪记录中都包含了完整的时间（Dates）和具体的地点（PdDistrict），这是我们预测犯罪的重要数据，下面对这类数据进行适当整理——转换为 0 或者 1。

```
In [19]: days = pd.get_dummies(crime_train['DayOfWeek'])
         hour = crime_train['Dates'].dt.hour
         hour = pd.get_dummies(hour)
         district = pd.get_dummies(crime_train['PdDistrict'])

         train_data = pd.concat([hour, days, district], axis=1)
         train_data['crime'] = crime
         train_data.head()
Out[19]:（输出结果如图 5-4-2 所示）
```

	0	1	2	3	4	5	6	7	8	9	...	CENTRAL	INGLESIDE	MISSION	NORTHERN	PARK	RICHMOND	SOUTHERN	TARAVA
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	1	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	1	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	1	0	0	0	
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	1	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	1	0	0	

5 rows × 42 columns

图 5-4-2　输出结果

（注：上述输出结果因为篇幅原因没有显示完整，请读者在调试程序时自行观察）

In[19]中使用了 Pandas 的 get_dummies()方法实现了 0/1 值的转换，方法示例请参考官方文档（https://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html）。

下面用同样的方法对测试集的数据进行整理。

```
In [20]: crime_test = pd.read_csv('/home/qiwsir/Documents/DataAnalysis/chapter05/
                                  crime_test.csv', parse_dates=['Dates'])
         days = pd.get_dummies(crime_train['DayOfWeek'])
         district = pd.get_dummies(crime_test['PdDistrict'])

         hour = crime_test['Dates'].dt.hour
         hour = pd.get_dummies(hour)

         test_data = pd.concat([hour, days, district], axis=1)
```

下面使用朴素贝叶斯方法的伯努利模型创建一个分类器。

```
In [21]: from sklearn.naive_bayes import BernoulliNB
         from sklearn.metrics import log_loss

         features = train_data.columns.values[:-1]

         training, testing = train_test_split(train_data, test_size=0.40)
         bl_model = BernoulliNB()
         bl_model.fit(training[features], training['crime'])
         test_pred = bl_model.predict_proba(testing[features])
         log_loss(testing['crime'], test_pred)
Out[21]: 2.5854488073215287
```

最后得到的是损失函数值，根据经验，这个结果还是可以的。不过，建议读者对比一下，运用前面的多项式模型再预测一下，同时计算损失函数——这是作业。

至此，笔者用几个示例简单讲解了朴素贝叶斯方法的应用，虽然是简单讲解，但读者也能够看到基本流程和方法。这些内容仅作为窥视机器学习的“小孔”，希望能够成为读者进入此领域的垫脚石。贝叶斯定理的应用不仅局限于上述范围，若要完整了解，请读者期待笔者后续作品，里面将系统地介绍贝叶斯定理。

跋

阅读到此处，虽然本书的内容就要结束了，但读者的数据分析师入门之路才刚刚开始。本书的所有内容，仅仅是入门或者是扫盲罢了。读者欲在数据分析领域有所作为，一方面需要补充知识，另一方面需要大量的练习，特别是在实战中练习。

在本书的叙述中，我经常建议读者查看帮助文档——不仅是本书，在“跟老齐学 Python”系列的任何一本书中，我都不厌其烦地如此建议。有的读者曾向我抱怨：“如果我自己能看文档，还来读你的书干什么？！”

此言有理！

读者不论是受本书启发，还是天资聪明，生来就晓得“查看帮助文档”，亦或从其他途径明白此道理，只要明白“看文档”的重要性，并且“肯于读文档”“能够读文档”，完全可以抛开本书。只是，达到如此境界，一般需要经过一个逐渐理解知识的过程，本书的作用就在于此。对于能够跨越发展的读者，本书的确多余了。

所以，阅读完本书之后，不论你觉得是否合算，也没法退货了。只要认识到并做到上面所述，就使用“精神胜利法”吧！

数据分析实战性强，不仅需要如本书所介绍的各种 Python 工具技能（本书所介绍的仅仅是基本的，还有很多更专业的工具），还需要数据对象所在行业的经验。因此，建议读者在学习之后，要尽可能与所在行业结合，思考是否能够用某个工具来优化某种工作流程，甚至创造一种新的产品或者服务，这样才能在入门的基础上有提升，甚至飞跃。

重复“序”中的那句话，“如果本书能够成为读者进入数据分析、机器学习领域的垫脚石，我当荣幸之至”。

世界是按某种规律创造的，我们有责任也有兴趣来寻找这个规律，数据分析就是我们手上的“神兵利器”。虽然它也有局限性，但至少在目前，我们还没有太讨厌它——更何况，会用它之后还可以有一个不错的饭碗。

先要有饭吃，再拯救世界。

希望本书能帮你达成第一个目标——有饭吃，吃好饭——成为吃货。

然后，请期待笔者后续的机器学习著作，那将是“诗和远方”了。

齐　伟

2018 年 3 月

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传　　真：（010）88254397

E-mail：　dbqq@phei.com.cn

通信地址：北京市海淀区万寿路 173 信箱

　　电子工业出版社总编办公室

邮　　编：100036